This is an example

You can type equations in-line like this $x = y$ and you can type them on their own line like this

$$x = y$$

You can also align equations like this

$$x = y$$
$$y = z$$

The stars used in the above align command make it so that there are no equation numbers. If you want to refer to an equation you can define it with the align environment (using no star) and a label

$$x = y \tag{1}$$

As you can see in Equation (1), we can refer to equations. You can highlight stuff like this. You can refer to citations by copying them into the file biblio.bib and then referring to them like this LeCun et al. [2010].

0.1 You can make subsections

Within a subsection, you can use headings such as

Example Here is an example of a paragraph heading.

You can use acronyms by adding them to the preamble_acronyms file and using them like this support vector machine (svm). Next time, it will appear like this: svm.

1 1/25/22 Introduction and Overview

skip

2 2/1/22 Supervised Learning

skip

3 2/8/22 A Bayesian Interlude

skip

4 2/15/2022 Regularization and Generalized Linear Models

This lecture talks about regularization as a concept and the different regularization techniques to prevent over-fitting. The last quarter of the class focuses on the use of the exponential family of distributions namely generalized linear models. Exponential families allows to skip through the tedious task of deriving a new model.

4.1 Regularization

As per the lecture the initial goal introduced was to understand how different genetic features play a role in the severity of a diseases. In order to achieve this goal we would would need to sample people from a distribution of those who have the disease. Along with this we need to take look into how many types of these genetic variants are present. Once this is done we need to weight these genetic variants to find which variant is more effective for the severity of the diseases. To solve this using math we, would use Linear regression model to minimize the least squares. The value of θ would be a good metric to see how important a particular feature is right ? But this isn't the scenario. Due to different scales of features, the value of θ value is simultaneously affected. In order to mitigate such a situation we need to standardize the features.

Linear Regression :

$$\min_{\theta} \frac{1}{n} \sum_{i=1}^{n} (\theta^T x_i - y_i)^2 \tag{2}$$

As seen the above case we have genetic variant features to be more than the data points that are sampled. This would lead to many solutions. We require a method by which we can consider/restrict the parameters(θ) of the model. That is to say when we do linear regression with the number of features that is greater than number of data points, we get lots of solutions given by:

$$\mathbf{X}\theta = \mathbf{y} \tag{3}$$

We achieve this restriction by using norms of the θ (norm -> distance between origin and the coefficient value). So by penalizing the norm(trying to bring the coefficients closer to 0) we will be able pay less importance to features that do not contribute much to the prediction/inference. We have different types of norm, namely Euclidean norm, Manhattan distance and Maximum norm.

Norm :

$$L(\theta) = \sum_{i=1}^{n} (\theta^T x_i - y_i)^2 + \lambda ||\theta||^2 \tag{4}$$

If we penalize the euclidean norm of the parameters it is called Ridge regression/L2 regression. The derivative of the loss function and equal it to zero the parameter matrix is of the following form :

$$\theta = (X^T X + \lambda I)^{-1} X^T y \tag{5}$$

The above equation gives use the characteristics of λ. As λ keeps increasing the values of coefficient will get closer to 0. In case of L2 it will never be 0. In order to make sure the penalty is not infinite we bound the norm penalty with a value M. Since we are using L2 regression technique all the parameters are bounded to a circle.

$$||\theta||_2^2 <= M \tag{6}$$

Example :

$$\theta_1^2 + \theta_2^2 = M \tag{7}$$

Which formulates to an equation of a circle. At an intuitive thinking L2 regularization penalizes steeper slope, which would over fit the data.

There is another type of norm available for regularization which is widely used for variable selection. This is the Manhattan distance. The regression model that uses this regularization type is called Lasso regression/ L1 regression.

$$L(\theta) = \sum_{i=1}^{n} (\theta^T x_i - y_i)^2 + \lambda ||\theta|| \tag{8}$$

A characteristic of L1 that differentiates from L2 is that when the norms are penalized near zero, the penalty is equal at all steps towards 0. So its heavily possible that there will be norms of coefficients that reach the origin (the coefficient values are 0). The parameters are constrained to a diamond around the origin.

The other regularization available is Elastic Net which combines L1 and L2. The Loss Function for the Elastic Net is expressed as:

$$\mathscr{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} (\theta^T x_i - y_i)^2 + \lambda(\alpha ||\theta||_1 + (1-\alpha)||\theta||_2^2) \tag{9}$$

Here α and λ are both hyperparameters and hence are not trainable since the gradient descent optimization will attempt to optimize the value of λ would go to zero and would eliminate the effect of regularization.

4.2 Analysis of Overfitting

Consider a problem where we aim to predict the heights of people given their diet. To do this, we must first gather data. This can be done by gathering information from people via surveys. Assuming $\mathscr{D}_1$ and $\mathscr{D}_2$ be the data gathered by two people performing the survey. $\mathscr{D}_1$ and $\mathscr{D}_2$ are random because we have not taken a survey of every person belonging to the $population$ but only a few people who form a $sample$ from that population. This way we can assume that the entire training data from the distribution of a random variable p. Assume that we have derived a hypothesis from the training set, i.e. our predictor is a function of the training set and we want to check its performance on the validation set. Let $\mathscr{D}$ be a dataset drawn from same distribution p which was used to generate the training set. Hence we solve for the minimum expected value of our hypothesis given as:

$$\min_f \mathbb{E}_{p(x,y),\mathscr{D}\sim p}[(y - f(x;\mathscr{D}))^2]$$

Conditioning the event on the input x we decompose the equation to include a variance term as follows:

$$\min_f \mathbb{E}_{p(x,y),\mathscr{D}\sim p}[(y - f(x;\mathscr{D}))^2] = \min_f \mathbb{E}_{p(x,y),\mathscr{D}\sim p}[(\mathbb{E}[y|x] - f(x;\mathscr{D}))^2] + \mathbb{E}_{p(x)}[\mathbb{V}ar(y|x)] \tag{10}$$

Solving for this equation results in three terms which are:

1. Bias: $\mathbb{E}_{p(x)}[(E[y|x] - \mathbb{E}_{\mathscr{D}\sim p}[f(x);\mathscr{D}])^2]$
2. Variance: $\mathbb{E}_{p(x)}[(\mathbb{E}_{\mathscr{D}\sim p}[f(x);\mathscr{D}] - f(x;\mathscr{D}))^2]$
3. Noise: $\mathbb{E}_{p(x)}[\mathbb{V}ar(y|x)]$

Description:

- The **bias** term represents the amount of predictions that differ from the actual values of the data set.
- The **variance** term accounts for finite data. It determines sensitivity of the model to the data set.
- The **noise** term represents accounts for y for not being a deterministic function of x.

Models that perform poorly on both the training and testing sets have high bias and high variance. These models tend to be smaller and suffer from *underfitting* on the training data. On the other hand, models that perform well on the training data set and poorly on the testing data set are said to have low bias and high variance. These models tend to be larger and suffer from *overfitting* on the data. An ideal model will have low bias and low variance. It is very difficult to minimize both the bias and variance at the same time. Hence we need to choose to minimize either one of them, i.e either the bias or the variance. Minimizing one may increase the other. This is also known as the *bias-variance tradeoff*.

4.3 Classification Problems

Suppose now that the response we are trying to predict is binary. That is, let $x_i \in \mathbb{R}^p$ and $y_i \in \{0, 1\}$ for $i = 1, \ldots, n$.

4.3.1 Maximum Likelihood Estimation

The *likelihood* of parameters (of a model) θ given a training example (x_i, y_i) is the probability of observing the data assuming that particular model. We denote this likelihood function by

$$\mathcal{L}(\theta|x, y) = P_\theta(x, y) = P_\theta(y|x)P(y).$$

(Here, θ parameterizes the conditional distribution only.) Intuitively, the best model we can find is the one that makes the data we observe the most probable. This insight leads us to a technique that can be applied to find the best parameters for a wide range of different kinds of models: *maximum likelihood estimation* (MLE).

Now let $X = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_n)$. If (x_i, y_i) are independent, then we can write the likelihood of the entire dataset as the product of the likelihoods for the individual points. We can express this idea as

$$\mathcal{L}(\theta|X, y) = \prod_{i=1}^{n} \mathcal{L}(\theta|x_i, y_i).$$

Because we are interested in maximizing $\mathcal{L}$, we can equivalently maximize $\log \mathcal{L}$. We can thus express the above relationship in terms of logs as

$$\ell(\theta|X, y) = \log \prod_{i=1}^{n} \mathcal{L}(\theta|x_i, y_i) = \sum_{i=1}^{n} \log \mathcal{L}(\theta|x_i, y_i) = \sum_{i=1}^{n} \ell(\theta|x_i, y_i).$$

At a high level, in order to estimate θ using maximum likelihood estimation, we carry out the following procedure:

1. Pick a distribution over the data points.
2. Write down the (log-)likelihood function for the distribution.
3. Maximize over θ using gradients.

4.3.2 Logistic Regression

We now introduce a maximum likelihood based method to solve a binary classification problem. In this case, we will assume that

$$y_i|x_i \sim \text{Bern}(\sigma(\theta^T x_i)),$$

where $\sigma(z) = (1 + e^{-z})^{-1}$ is a function from $\mathbb{R}$ into $(0, 1)$ (which we can interpret as probabilities). We can reformulate this as

$$p(y_i = 1|x_i) = \sigma(\theta^T x_i)^{y_i}(1 - \sigma(\theta^T x_i))^{1-y_i}.$$

Now, we can write down the log-likelihood function for a particular parameter θ

$$\ell(\theta|X, y) = \sum_{i=1}^{n} y_i \log \sigma(\theta^T x_i) + (1 - y_i) \log(1 - \sigma(\theta^T x_i)) + \log P(y_i). \tag{11}$$

To maximize over values of θ, we take derivatives with respect to θ, using the fact that that $\sigma'(z) = \sigma(z)(1 - \sigma(z))$.

$$\nabla_\theta \ell(\theta|X, y) = \sum_{i=1}^{n} \frac{y_i x_i}{\sigma(\theta^T x_i)} \sigma(\theta^T x_i)(1 - \sigma(\theta^T x_i)) - \frac{(1 - y_i)x_i}{1 - \sigma(\theta^T x_i)} \sigma(\theta^T x_i)(1 - \sigma(\theta^T x_i)) \tag{12}$$

$$= \sum_{i=1}^{n} x_i (y_i (1 - \sigma(\theta^T x_i)) - (1 - y_i)\sigma(\theta^T x_i)) \tag{13}$$

$$= \sum_{i=1}^{n} x_i (y_i - y_i \sigma(\theta^T x_i) - \sigma(\theta^T x_i) + y_i \sigma(\theta^T x_i)) \tag{14}$$

$$= \sum_{i=1}^{n} x_i (y_i - \sigma(\theta^T x_i)). \tag{15}$$

Due to the functional form of $\sigma(z)$, there is no analytical solution, but we can use stochastic gradient ascent to find θ from here.

One obvious question that we have not addressed is how we made our distributional assumption at the very beginning before we turned the maximum likelihood crank. To give some intuition, we note that $\sigma(z)$ is actually the inverse of the log-odds function $\text{logit}(p) = \log(p/(1-p))$. This reveals an important fact: logistic regression is linear in the log-odds, i.e., $\text{logit}(p(y=1|x)) = -\log(\exp(-\theta^T x)) = \theta^T x$. Whereas in linear regression, the parameters indicate the change in response we expect per change in the corresponding feature value, in logistic regression, the parameters indicate the change in the log-odds that would occur in response to a change in the corresponding feature values. Finally, we note that overfitting can be addressed with the same regularization techniques described earlier.

4.4 Generalized Linear Models (GLMs)

So far, we have seen various derivations of MLE for different distributional assumptions (Bernoulli, Normal). Is there a more general framework we can use to think about distributions that will save us the work of re-deriving everything from scratch every time?

4.4.1 Exponential Families

Enter exponential families! An exponential family distribution is one of the form

$$p(x) = h(x)\exp(\eta^T t(x) - a(\eta)),$$

where h is the base measure, η is the model parameter, $t(x)$ is a sufficient statistic, and $a(\eta)$ is what's called the log-normalizer. The log-normalizer is given by

$$a(\eta) = \log \int h(x)\exp(\eta^T t(x))dx.$$

A very useful property of the log-normalizer is that

$$\nabla a(\eta) = \int t(x)h(x)\exp(\eta^T t(x) - a(\eta))dx \tag{16}$$

$$= \int t(x)p(x)dx \tag{17}$$

$$= \mathbb{E}[t(x)]. \tag{18}$$

4.4.2 A generalized linear model (GLM)

A generalized linear model or GLM is a conditional version of an exponential dispersion family distribution, in which the natural parameters are a linear function of the input.

More precisely, the model has the following form:

$$p(y_i|x_i) = h(y_i)\exp(\eta^T t(y_i) - a(\eta)),$$

one option

$$\eta_i = \theta^T x_i,$$

And we learn parameters by maximizing likelihood

$$\mathcal{L}(\theta) = \sum_i log(h(y_i)exp(\eta_i^T t(y_i) - a(\eta_i)))$$

We now look at how we can recast the problem of finding optimal parameters for logistic regression through the lens of GLMs.

4.4.3 Logistic Regression via GLMs

If $z \sim \text{Bern}(\sigma(\eta))$, then we have

$$p(z) = \sigma(\eta)^z(1-\sigma(\eta))^{1-z} \tag{19}$$
$$= \exp(\log(\sigma(\eta)^z(1-\sigma(\eta))^{1-z})) \tag{20}$$
$$= \exp(z\log(\sigma(\eta)) + (1-z)\log(1-\sigma(\eta))) \tag{21}$$
$$= \exp(z\log(\sigma(\eta)/(1-\sigma(\eta))) + \log(1-\sigma(\eta))) \tag{22}$$
$$= \exp\left(z\sigma(\eta) + \log\left(\frac{\exp(-\eta)}{1+\exp(-\eta)}\right)\right) \tag{23}$$
$$= \exp\left(z\sigma(\eta) + \log\left(\frac{\exp(-\eta)}{\exp(-\eta)}\left(\frac{1}{\frac{1}{\exp(-\eta)}+1}\right)\right)\right) \tag{24}$$
$$= \exp(z\sigma(\eta) - \log(1+\exp(\eta))). \tag{25}$$

We can thus see that the Bernoulli distribution is indeed an exponential family, with

$$t(z) = z \tag{26}$$
$$h(z) = 1 \tag{27}$$
$$a(\eta) = \log(1+\exp(\eta)). \tag{28}$$

Thus, if we model $y_i|x_i \sim \text{Bern}(f(\eta_i))$ for some function f, then using what we know about exponential families, we can actually find $\mathbb{E}[y_i|x_i]$ by differentiating the log-normalizer with respect to η_i. Doing so, we see that

$$\mathbb{E}[y_i|x_i] = p(y_i = 1|x_i) = \nabla_{\eta_i}\log(1+\exp(\eta_i)) = \sigma(\eta_i),$$

so $f = \sigma$. If we set $\eta_i = \theta^T x_i$, we see that a GLM with a Bernoulli distribution is exactly logistic regression!

More generally, if $\eta_i = \theta^T x_i$, we can estimate the parameters θ by maximizing

$$\ell(\theta|X,y) = \sum_{i=1}^{n}\log(h(y_i)) + \eta_i^T t(y_i) - a(\eta_i).$$

We can drop the first sum and because it does not depend on θ, so we have

$$\theta^\star = \text{argmax}_\theta \sum_{i=1}^{n}\eta_i^T t(y_i) - a(\eta_i).$$

Just to check, we note that for logistic regression, we have

$$\theta^\star = \text{argmax}_\theta \sum_{i=1}^{n}(\theta^T x_i)y_i - \log(1+\exp(\theta^T x_i)),$$

which, with some algebra, we can show is exactly the same as the log-likelihood we derived in 11.

5 2/22/2022 Neural Networks

This lecture discussed Neural Networks, specifically, an analysis of fully connected networks, convolution neural networks, their properties, and their generalization capabilities.

5.1 Space Transformations

In this section, we get the intuition about how linear and non-linear transformations effect a given random cloud of points.
Here the cloud of points are coming from a 2-D Gaussian with $\sigma = 1$.

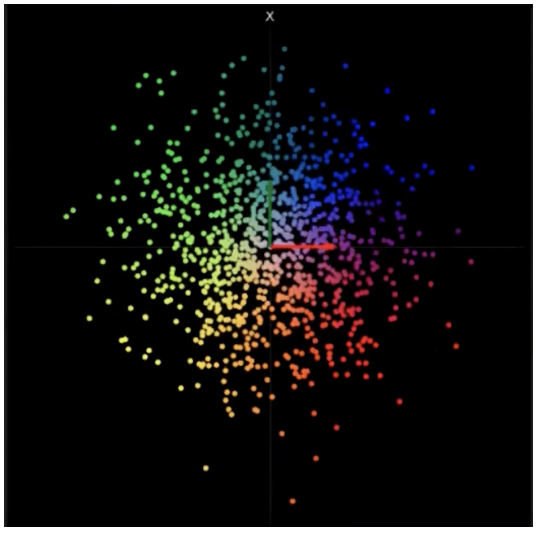


Figure 1: 2D Gaussian with std=1

After applying linear transformation to cloud we get the below picture

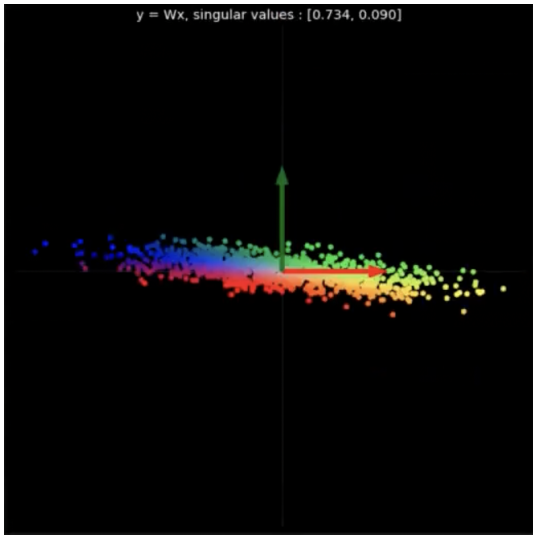


Figure 2: Linear Transformation on Fig:1

We observe that with linear transformation we can $\boldsymbol{Rotate}$, $\boldsymbol{Reflect}$, $\boldsymbol{Stretch}$ and $\boldsymbol{Compress}$.

With the linear transformation we cannot curve a function. To do this, we need to use non-linear function. Now if we apply a non-linear function to the given Fig: 1, we get almost uniform distribution.

We can approximately map points to a square by first stretching out by a factor s and then squashing with tanh function.

$$f(x) = \tanh\left(\begin{bmatrix} s & 0 \\ 0 & s \end{bmatrix} x\right)$$

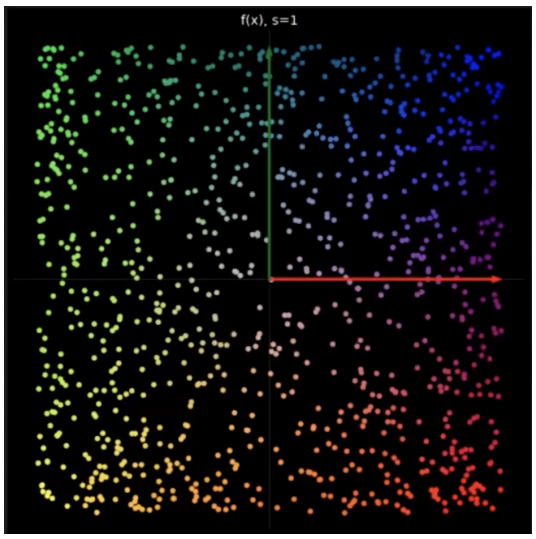
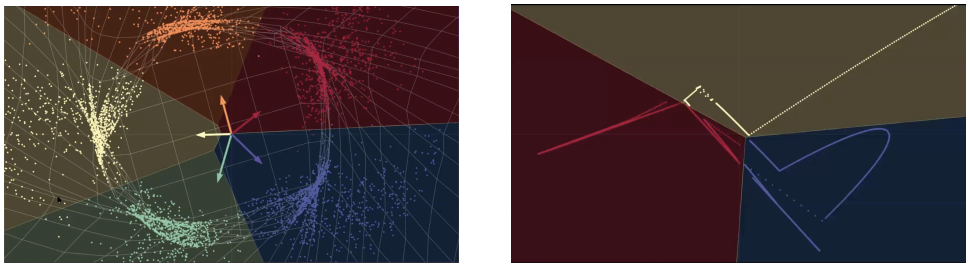


Figure 3: Non-Linear Transformation on Fig:1

Finally we see how a model moves the data such that a linear classifier could classify it. Below we have two images which show how a model classifies data in high dimension (left side image) and low dimension (right side image).
The left side image has hundreds of neurons with many hidden layers and easy to train. While the right side image was a skinny network with few neurons and only 1 hidden layer with 100 of units. The **major comparison** we have here is that the model with less dimension is harder to train.



(a) High Dimension **(b)** Low Dimension

Figure 4: High and Low Dimension Classification

5.2 Neural Networks

Neural networks comprise of a composition functions, which may take the following forms:

1. Affine Functions: These result in transformations of the input, for example functions of the form
$$\boldsymbol{h} = \boldsymbol{W_h x} + \boldsymbol{b_h}$$
2. Non-linear Functions: These ensure that the network does not collapse to simply a linear model. Commonly used examples are the hyperbolic tangent and sigmoid functions.

3. Linear Functions: These are used to easily change the size of an input or intermediate matrix or vector.

Linear models can be used for classification, however, mis-classification occurs with linear classifiers when training points of a particular class cross the boundary of the classifier. A simple solution to this is to move points around so that they can be classified without error. This can be easily done using *transformations* through neural networks.

Specifically, *hidden units* are used to move the input from the input space to an intermediate space, to obtain a high-level representation of the target. The dimensionality of these hidden layers govern the dimension to which the input is mapped to (in that specific layer). Generally, a *skinnier* network (with fewer neurons) may be harder to train than a shallower but denser network.

Before discussing how Neural Network Inference works, we give a brief overview on the observability of the data. Typically, for a neural network, the data has three main components, which we represent by x, y, and z.

- x is the input, which is fully observable
- y is the target, which may be observable (in the case of *supervised* learning) or unobservable (in the case of *unsupervised* learning)
- z is the latent variable, which is always hidden

5.2.1 Neural Network Inference

The pipeline for inference in a neural network is as follows:

1. The input **x** is passed through a *predictor*, which moves the input from the x-space towards the y-space, resulting in a hidden representation **h**. **h** is a high level representation of the target, who's dimensionality is guided by the number of hidden units. Specifically,

$$\boldsymbol{h} = f(\boldsymbol{W_h x} + \boldsymbol{b_h}) \tag{29}$$

   Where, f is a non linear function that takes in an affine transformation of $\boldsymbol{x}$.

2. The hidden representation **h** is then passed through a *decoder*, which moves it from the h-space to the y-space, resulting in a prediction $\tilde{y}$. Here,

$$\tilde{y} = g(\boldsymbol{W_y h} + \boldsymbol{b_y}) \tag{30}$$

   Where, g is a non linear function that takes in an affine transformation of $\boldsymbol{h}$. For Classification problems, the prediction $\tilde{y}$ is commonly represented by a one-hot encoding (or a soft one-hot encoding) or argmax (softargmax) of dimension C (the number of classes).

5.3 Energy, Loss and Cost Functions

The computation of a neural network's loss, cost, and energy determine its performance. **Loss** indicates how bad a specific parameterization is for the training set. The system's **energy** is the measure of the incompatibility between an observation or input x and a target y. In the case of neural network inference, we will consider the system's energy to be *equal* to the cost.

This equation of energy to cost begs the question: What is the cost of a system, and why is it important? **Cost** is the distance between the target y and the predictor $\tilde{y}$. Since neural networks use gradient-based learning, which involves computing the cost's gradient, choosing the right cost function is important. If the model is well-trained, we expect $\tilde{y}$ to be close to y, which leads to a reduction in the cost and, in turn, the energy of the neural network model. Lower cost(and energy) indicates that the observation x and target y are compatible. If the model is well trained, the cost increases if the prediction gives an absurd value or the target changes.

At this point, another question arises: is the incompatibility between an input and a target dependent on the model? The answer is yes. A model's energy is dependent on the model itself. If the model is badly trained, its predictions will not be reliable, and the input and target will be incompatible.

One example of a cost function is the negative log of the inner product of the target and prediction $-\log y^T \tilde{y}$, which is also called the cross-entropy. Another example is a functional or a mapping from functions to real numbers whose minimum occurs at a specific function.

5.4 Gradient Descent and Backpropogation

We solve and model the performance of machine learning problems generally in terms of the cost function. It is a function of weights (parameters). In the feature (weight) space, at a particular point in that space, the loss will have some value (loss) which is a function of weights. This loss value is a measure of how bad the values of the current weight vector is on the training set which means lower the value better the weight vector is, and the better that weight vector represents our training data. So the goal of a learning algorithm is to minimize the loss function in the feature space i.e., choose such a vector of weights that best represents our training data. Just choosing a feature vector randomly and checking loss value in hope that we might end up with a weight vector that has a low loss value wouldn't work. A good idea would be to calculate the gradient of the function with respect to weights. This gives us the direction in weight space which maximally increases the value of loss function. Is this our goal? Of course not! The goal of the loss function is to have as low a value as possible for a weight vector. So a simple solution is to choose the direction exactly opposite of that of the gradient. Hence, it gives the direction which maximally decreases the value of loss function at that particular point (a.k.a our goal). So, what we do is we take a step in that direction which ends up reducing the value of loss function. The exact length of the step is determined by the learning rate which is just a value multiplied by the gradient. Therefore, gradient together with learning rate determines which direction and how much should weights be changed such that loss is minimized than its current value. If this process is repeated multiple times, it is expected that we end at a feasible solution point i.e., where values of weights are such that it represents the training data perfectly (where the loss value is minimum).

Let's try to understand this with an example. The loss discussed in the previous section tells us about the incompatibility between the parameters and the training set. It tells us how bad our choice of parameters is with respect to the training set. So we try to get something better by training the network.

$$w = \{W_h, b_h, W_y, b_y\}$$

Let w be a collection of all the possible training parameters of the network. We will have a set of loss function $\mathscr{L}(w, S) \in \mathbb{R}^+$ which tell us about how bad the parameters are.

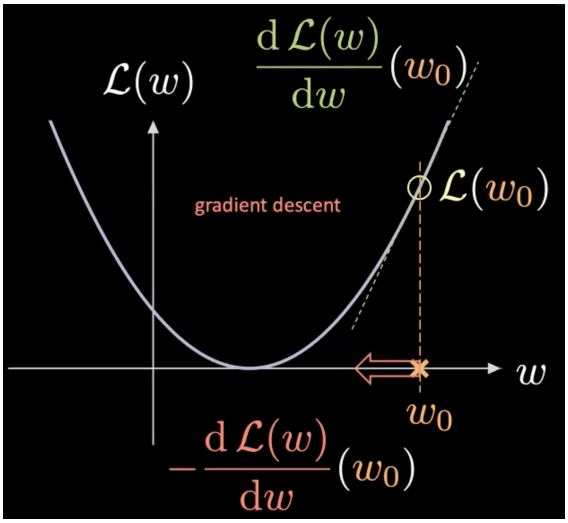


Figure 5: Gradient Descent

Assume that we have only one dimensional weights. In this case, the loss is quadratic. We take a random initialization w_0 and we evaluate the value of loss at that point as can be seen in the example in Figure 5. The loss at w_0 is pretty high which means that the network is very bad. Now, we calculate the derivative at that point. We see that the derivative is positive, so we move towards the left i.e. multiply w_0 with the negative derivative. This is gradient descent. The gradient will basically tell us

what is the direction in which we have the steepest ascent and we are taking a step in the opposite direction to go down. So, gradient descent tells us about how to move in the parameter space in order to lower the badness of the parameters so that the task at hand is performed better. How do we do that?

We just perform the chain rule which in deep learning is called Back Propagation (Figure 6). It is used for computing the gradients. Gradient descent is used to update the weights during the training in our case. It is not just used for training, but can also be used for inference.

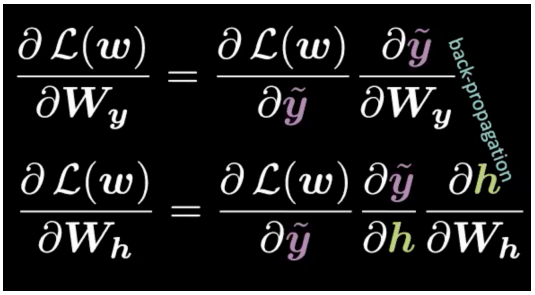


Figure 6: Back Propagation

5.5 Convolution Neural Networks

As we've seen above, fully connected networks are really powerful and can even perfectly learn the noise in the training set. They are like a very large parametric function. We would like to restrict the network by limiting the number of parameters. Natural signals have a peculiar property which we exploit. We now consider natural signals as our inputs which can be represented by the following set of functions:

$$\chi = \{x^i : \Omega \to \mathbb{R}^c, \omega \mapsto x^i(\omega)\}_{i=1}^m \tag{31}$$

where Ω is the domain (like time in case of audio or length and width in case of an image) and c is the number of channels in the input layer. These signals can be of single or multiple channels. For example, an audio signal can be monophonic (single channel) or stereophonic (dual channels). A greyscale image has a single channel, an RGB image (colored) has 3 channels. So now these signals can be represented as vectors.

Such signals have some interesting properties.

- **Stationarity**: Similar patterns reoccur multiple times over the input signal.
- **Locality**: Points in close intervals are related. Farther the points are, lesser related are they.
- **Compositionality**: Information from data is present at different resolutions.

Fully connected networks have way too many parameters to be trained. Some real world signals like images already have huge input size. Connecting such input to a larger hidden layer in such a network would tremendously blow up the number of parameters.

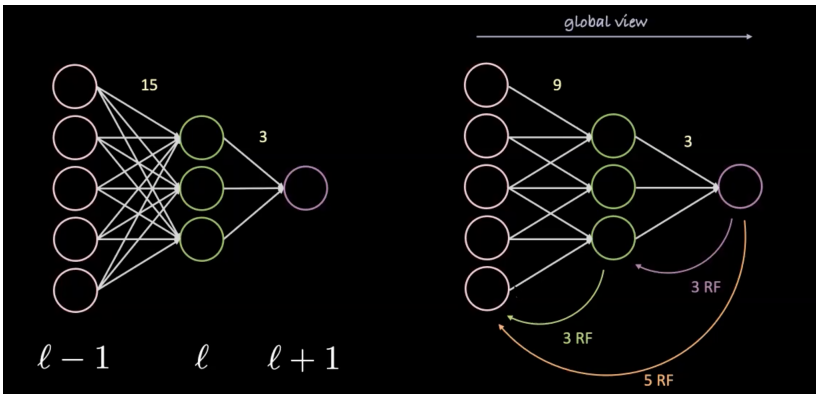


Figure 7: Fully Connected Network (on left) vs Convolutional Neural Network (on right)

In a convolutional neural network, we have connections from only some points in a close interval (instead of all points in the layer) to a unit in the next layer. Thus the total number of parameters are greatly reduced, yet capturing the information in the input by exploiting the properties of such signals. **Receptive field** is defined as the total space construct including units that offer input to the next layer's units.

Since we have reoccurring patterns across independent domains in these signals, we can have the same values of the parameters being multiplied at different portions of a domain, to capture similar type of data from them. Hence by **Parameter Sharing**, we reduce the number of parameters to be trained by a huge factor.

CNN has multiple benefits over fully connected networks for such data. We get faster convergence as much more data is used to determine the values of these parameters. Making the use of locality of features, we get a better generalisation. Unlike fully connected network, CNN is not constrained to the input size. Due to **kernel** independence, we get the ability for high parallelisation. The total number of parameters are much less. So connection sparsity reduces the amount of computations.

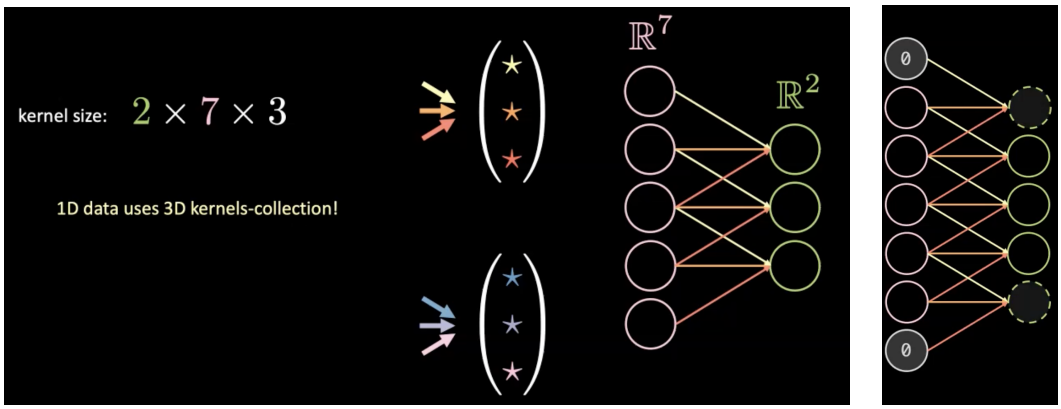


Figure 8: Kernels (on left) and Padding (on right) in CNN

We obtain one scalar value from each kernel at any unit in the output layer. So the dimension of each unit equals the number of kernels that are used to compute its value. Dimension of a unit in a kernel equals the dimension of a unit in the input layer.

We add zero units in the input layer at the boundaries which is known as **padding** so that the size of each channel in the output layer is the same as that of the input layer.

5.6 Generalization

We discuss generalisation of models - which means how the model predicts when new data (sampled from the same train distribution) is presented to the model. The model tries to predict an estimate of an unseen data input based on the learnings it has from seen input data. This is also called as Inductive Bias.

5.6.1 Generalisation error

To define Generalisation Error we can consider an example: if a model is unable to predict accurately on unseen data, then the model has a high generalisation error. The Figure 9 shows the trend of Generalisation error with respect to model complexity. Note: Y-axis shows: Incompatibility of weights with the test set. When the model-complexity is low, which means the model is too simple to learn trends in the data. This shows how the $variance$ is low and $bias^2$ is too high. This would result in high generalisation error. Similarly when the model complexity increases, the $variance$ increases because the model tries to fit the noise in the dataset even though the bias has reduced. We see a dip in the generalisation error when we go from a very simple model to a complex model - but generalisation quickly changes increases when the model gets too complex.

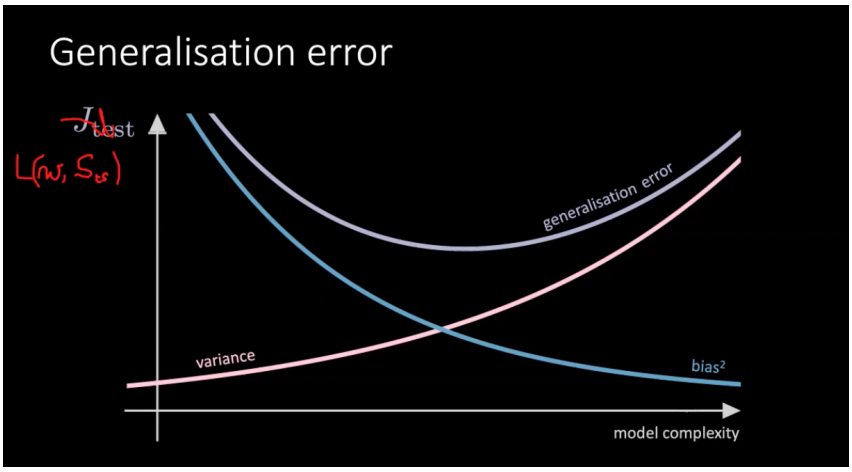


Figure 9: Generalisation error; y:axis: Incompatibility of Model, x:axis: Model complexity

5.6.2 MLP vs CNN

Let's train multiple models on different subsets of dataset to reach a target for a fully connected Neural Network(MLP) and for a CNN. We observe that average of multiple models in both the neural network layers result in intersecting with the target. But, we see in the case of MLP(Fully connected Neural Network) as the parameter space is large, the variance (average square distance to the average of a model type) is larger than compared to varaince of CNN. The variance of two models are represented in the Figure 10 with the dotted circles. The variance of CNN is smaller because CNN has a smaller search space compared to a Fully connected Neural Network(MLP).

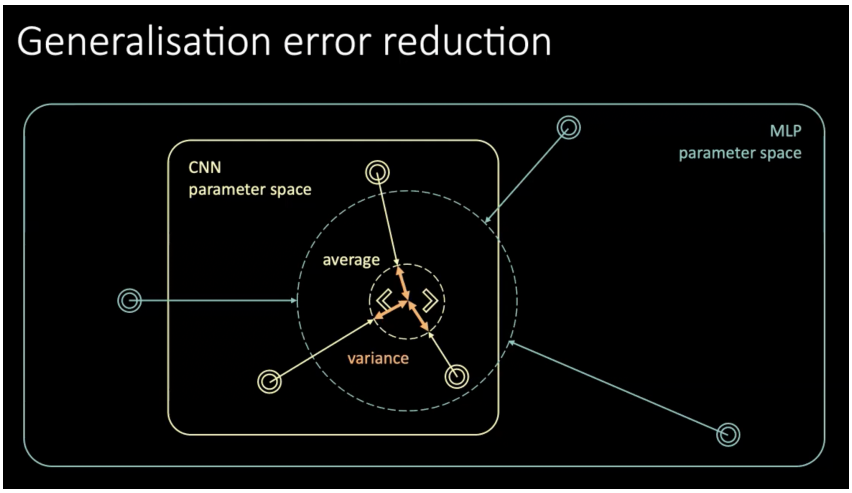


Figure 10: Generalisation Error reduction in MLP vs CNN

6 3/1/2022 Trees and Forests

This lecture discussed Machine Learning beyond Linearity, Decision Trees, Random Forests, and Gradient Boosting.

6.1 ML Beyond Linearity

Let's say that the goal is to understand the role of Vitamin D on health. How can we approach it?

1. We observe the data $(x_1, y_1)...(x_n, y_n)$
2. There is a positive correlation.
3. What would the Generalised Linear Model say?

The positive correlation indicates that more Vitamin D is better. The inference is to give ∞ Vitamin D to people. In this small example, we have seen that the plausibility of linearity is just very low. Deploying this linear model would result in everyone getting giant amounts of Vitamin D which is also bad for health (hypercalcemia).
The linear models are limited. The relationships will always be positive, negative, or 0. The inference is that too little Vitamin D is bad, too much is also bad, and hence the linearity does not fit.
Linear models can be non linear:

$$y_i = \theta_1 x_{i1} + \theta_0 + \epsilon_1 \tag{32}$$

We can make the above equation non-linear by adding more features that are functions of existing features. This is shown in the below equation:

$$y_i = \theta_1 x_{i1} + \theta_2 x_{i1}^2 + \theta_0 + \epsilon_1 \tag{33}$$

We have now introduced a quadratic function of x_1 but this is still linear in the new feature space. We can learn about the parameters by generalising the model.

Convert Linear to a non-linear model

1. Add higher order features.
2. Estimate the parameters of augmented model having extra features inside of it.

If we add some extra facts, for example, Vitamin K influences blood calcium. Now we know that Vitamin D influences it too, so we have to factor that in. So, our new goal is to understand the influence of Vitamin D and K on health.

$$y_i = \theta_1 x_{ivit-D} + \theta_2 x_{ivit-K} + \theta_0 + \epsilon_i \tag{34}$$

The linear model with Vitamin D and Vitamin K predicts the health. To handle the non-linearity of Vitamin D, we add one more feature which is shown in the equation below:

$$y_i = \theta_1 x_{ivit-D} + \theta_3 x_{ivit-D}^2 + \theta_2 x_{ivit-k} + \theta_0 + \epsilon_i \tag{35}$$

But, how do we calculate how Vitamin D and K react together? For example, too much Vitamin D and little Vitamin K might be OK as K might counter D). We try to include the interaction between them below:

$$y_i = \theta_1 x_{ivit-D} + \theta_3 x_{ivit-D}^2 + \theta_2 x_{ivit-k} + \theta_4 x_{ivit-D} x_{ivit-K} + \theta_0 + \epsilon_i \tag{36}$$

This model can still be trained by the Generalised Learning Model (GLM). We can convert the linear to non-linear model by precomputing the feature combination and training the GLM. There are a few disadvantages:

1. We do not know when to stop. We can keep on increasing the power of x.
2. This is like the chicken and the egg problem.
3. We need some enumeration to add features which could be expensive.
4. Scales very poorly with dimensionality.
5. Preconceived relation between the features should be known.

We want models with non-linearities that take in raw features and tell us based on labels and training data, what functions should be used.

6.2 Decision Trees

1. We divide feature space into many regions.
2. We have different labels/predictions in each region.
3. Performs classifications or regression.

$$pred(x^*) = \sum_{i=1}^{N_{leaves}} \mathbb{1}[x^* \in leaf_i]\mathbb{E}[y|x \in leaf_i] \tag{37}$$

Where $leaf_i$ is the intersection of sets along the path to the leaf

Learning a Decision Tree

1. Is there an optimal tree giving optimal performance.
2. It is a combinatorial process that is, we can pick from a collection of features at every level.
3. We can use the greedy algorithm which will give the best choice locally.

Greedy Algorithm:

1. We figure out a rule to split the input space.
2. Split the space according to the split above.
3. Start over again in each new subspace. The Basic idea of Greedy Tree Building is shown in Figure 11

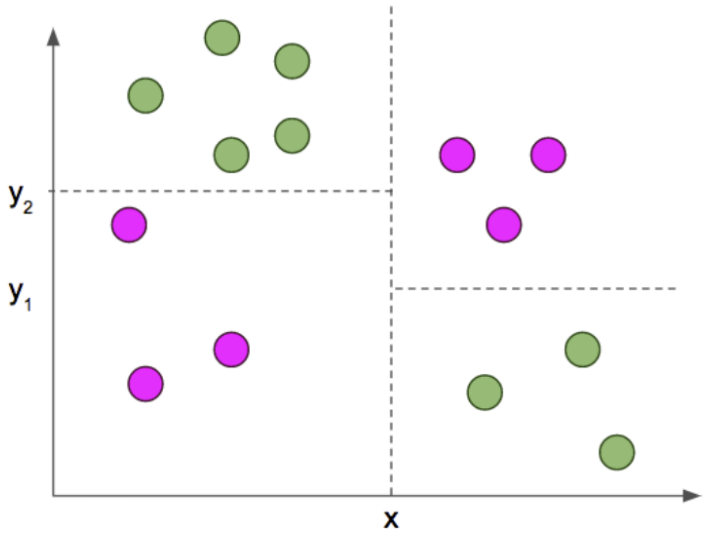


Figure 11: Greedy Tree Building

Classification Tree: In this, we assume that all features x are continuous. It can work with discrete variables too if we split on different categories. For every feature j, we find a split point c_j to minimise classification error in the new subsets that are formed.
The prediction value is given by:

$$prediction(A_j) = majority(x_j \in A_j) \tag{38}$$

The error on split is given by:

$$min_{c_j} \frac{1}{n} \sum_{i=1}^{n} \mathbb{1}[prediction(x_{i,j} \leq c_j) \neq y_i] \tag{39}$$

Different Choices for Information Measures:

1. Entropy: Measure of unpredictability.
2. Mutual Information: If y is a binary variable, x will be split into binary variable based on split point we choose. We need to find the split value.
3. KL Divergence: Based on the choice of likelihood, we might want to find a θ to minimise the KL Divergence.

Other Types of Error:

1. Conditional Entropy: We want to pick a c_j such that there is low uncertainty about the label. We want this value to be small.
2. Mutual Information: We want to find a variable that has high information with label y.

$$MI(y; a_j) = \mathbb{E}_{p(a_j,y)}\left[log\frac{p(a_j, y)}{p(a_j)p(y)}\right] \tag{40}$$

   If a_j and y are independent, MI = 0.

We can use the same idea in regression if we find splits that minimize the regression error in each split. **What happens if we keep splitting:**

1. Overfitting.
2. It will not generalise well.
3. We can fix this by:
   (a) Requiring multiple data points in each leaf.
   (b) A test for significance in the improvement.
   (c) Pruning the trees using the test data. We may have a validation set and if we train a really big tree, we can work from bottom and check at what level the errors are coming and prune from there.

Sensitive Greedy algorithms:

1. Small changes in the data can create different predictions.
2. If two features are similar? If we happen to observe $1/2$ eggs that make the features better, we will split on that one first. It helps us take better decisions.
3. When data in each region gets small. We still need to be careful about overfitting.

We want a way to learn trees that are less prone to overfitting and robust to perturbations in the data. We don't want an algorithm such that we change 1 example and end up getting a completely different tree. Hence, we can split on combination of features!
We can average lots of randomly perturbed trees together as that would be robust to perturbations by averaging over them and adding more trees cannot overfit.

6.3 Random Forests

We start by building a random tree $h(x, \theta_k)$ where θ_k is sampled from a distribution q, which depends on the training data. $(\theta_k \sim q)$ This tree outputs some value based on the input x. We draw randomly from a collection of features and split the tree on that one. The tree parameters θ_k are the splits or split values. We build similar trees but from different data samples. The features are randomly selected during their construction. This leads to the creation of random forests. The parameter θ_k can be thought of as an ordering of the splits. This leads to the question- *How many trees can we construct?* - Well, we can build as many trees as our computer can support. Adding more trees won't decrease the model efficiency.

The Generalisation Error:
This gives us a measure of how well this model would perform when we have a new data sample. It is the error based on the label vs prediction of a sample. Suppose we have n data points: $(x_1, y_1)\ldots(x_n, y_n)$ sampled from p. Let f be the model we train on this data. We want to minimise

the error over all our data points, $min_f Error[(x_1, y_1)\dots(x_n, y_n)]$. The generalisation error of f is given as:

$$E_{(x^*,y^*)\sim p}[Error(y^*, f(x^*))] \tag{41}$$

The generalisation error tells us how good the function f is for predicting y. To evaluate the generalisation error, we can use the training set. Let D be the data collection of n points. We can hold some validation/test set and try to estimate the error against this set. Now, we have 2 samples for p- the training collection D and the single test sample. The generalisation error (ge) of f is given as:

$$\mathop{\mathbb{E}}_{D=(x_1,y_1)\dots(x_n,y_n)\sim p} \mathop{\mathbb{E}}_{(x^*,y^*)\sim p} [Error(y^*, f(x^*; D))] \tag{42}$$

Using the same data in both the expectations above is not unbiased expectation. It is a mechanical way of seeing what over fitting would be (when the train error is much lower than the test).

The Margin:
The margin of a forest is defined as the fraction of trees that are correct $\sum_{k=1}^{K} 1[h(x;\theta_k) = y]$ minus those that are incorrect $\sum_{k=1}^{K} 1[h(x;\theta_k) = 1-y]$.

$$margin(x, y) = \frac{1}{K}\sum_{k=1}^{K} 1[h(x;\theta_k) = y] - \frac{1}{K}\sum_{k=1}^{K} 1[h(x;\theta_k) = 1-y] \tag{43}$$

When the margin is lesser than 0, it implies that majority of trees in the forest are performing incorrectly. The generalisation error is hence an expectation of the margin for a new data sample being less than 0.
If all trees are sampled from the same distribution, adding more trees will result in the ge converging to the expectation. It does not harm the generalisation. This stands true only as long as we do not change the sampling distribution. The ge in terms of the margin of the forest is given as:

$$ge = E_{(x,y)\sim p}[E_{\theta\sim q}[1[h(x;\theta) = y] - 1[h(x;\theta) = 1-y]] < 0]$$
$$\implies ge = P_{(x,y)}[margin(x, y) < 0]$$

Now, let us take a moment to revise 2 inequalities we have studied before:

1. Chebyshev's Inequality: How far can a random variable deviate from its mean and absolute value. Specifically, no more than $\frac{1}{k^2}$ of the distribution's values can be k or more standard deviations away from the mean.
   $P(|x-\mu| \geq k\sigma) \leq \frac{1}{k^2}$
2. Jensen's Inequality: It states that the convex transformation of a mean is less than or equal to the mean applied after convex transformation. Let x be a random variable and f be a convex function.
   $\mathbb{E}[f(x)] \geq f(\mathbb{E}[x])$

Now, we define the strength s of a forest as its expected margin. $s = \mathbb{E}_{x,y}[margin(x, y)]$. Using the properties of Chebyshev's and Jensen's inequalities, $P[A \leq -b] + P[A \geq b] = P[|A| \geq b]$ and $k\sigma = \mathbb{E}[margin]$, we get:

$$\begin{aligned}
ge &= P_{(x,y)}[margin(x, y) < 0] \\
&= P_{(x,y)}[margin(x, y) - \mathop{\mathbb{E}}_{x,y}[margin(x, y)] < -\mathop{\mathbb{E}}_{x,y}[margin(x, y)]] \\
&\leq P_{(x,y)}[margin(x, y) - \mathop{\mathbb{E}}_{x,y}[margin(x, y)] \leq -\mathop{\mathbb{E}}_{x,y}[margin(x, y)]] \\
&+ P_{(x,y)}[margin(x, y) - \mathop{\mathbb{E}}_{x,y}[margin(x, y)] \geq -\mathop{\mathbb{E}}_{x,y}[margin(x, y)]] \\
&= P_{(x,y)}[|margin(x, y) - \mathop{\mathbb{E}}_{x,y}[margin(x, y)]| \geq \mathop{\mathbb{E}}_{x,y}[margin(x, y)]] \\
&\leq \frac{\mathbb{V}ar[margin]}{\mathbb{E}[margin]^2} = \frac{\mathbb{V}ar[margin]}{s^2}
\end{aligned}$$

The two inequalities bring about a certain degree of sloppiness in the equations above. There are 2 sources of randomness in a forest- the sampling distribution of the trees and the test data point. We get an intuition that the distribution that produces trees as a forest having high strength will result in good generalization. However, we get no intuition for the variance and will have to inspect it further. To break the variance, we first write an identity showing the expectation of the square of a r.v. can be written as double expectation.

$$\mathop{\mathbb{E}}_{\theta\sim q} h(\theta)^2 = \mathop{\mathbb{E}}_{\theta,\theta'\sim q}[h(\theta)h(\theta')] \tag{44}$$

We now define the raw margin as the margin for a single tree with fragmented data.

$$\begin{aligned} margin(x,y) &= \mathop{\mathbb{E}}_{\theta\sim q}[1[h(x;\theta)=y]-1[h(x;\theta)=1-y]] \\ &= \mathop{\mathbb{E}}_{\theta\sim q}[raw-margin(x,y,\theta)] \end{aligned}$$

From the above two, we get the square of the margin as the expectation of product of raw-margins of 2 trees:

$$margin(x,y)^2 = E_{\theta,\theta'\sim q}[raw-margin(x,y,\theta)raw-margin(x,y,\theta')]$$

So, now we have

$$\mathbb{V}ar[margin] = \mathop{\mathbb{E}}_{x,y\sim p}[(margin(x,y)-\mathop{\mathbb{E}}_{x^*,y^*\sim p}[margin(x^*,y^*)])^2]$$

(Substituting the raw-margin above)

$$= \mathop{\mathbb{E}}_{x,y\sim p}[(\mathop{\mathbb{E}}_{\theta\sim q}[rm(x,y,\theta)]-\mathop{\mathbb{E}}_{x^*,y^*\sim p}[\mathop{\mathbb{E}}_{\theta\sim q}[rm(x^*,y^*,\theta)]])^2]$$

(Swapping order of expectations)

$$= \mathop{\mathbb{E}}_{x,y\sim p}[(\mathop{\mathbb{E}}_{\theta\sim q}[rm(x,y,\theta)]-\mathop{\mathbb{E}}_{\theta\sim q}[\mathop{\mathbb{E}}_{x^*,y^*\sim p}[rm(x^*,y^*,\theta)]])^2]$$

(Combining quantities since expectation is linear)

$$= \mathop{\mathbb{E}}_{x,y\sim p}[(\mathop{\mathbb{E}}_{\theta\sim q}[rm(x,y,\theta)-\mathop{\mathbb{E}}_{x^*,y^*\sim p}[rm(x^*,y^*,\theta)]])^2]$$

(Expectation of rm squared on the outside results in double expectation of the rm multiplied by the rm of a different tree)

$$= \mathop{\mathbb{E}}_{x,y\sim p}[\mathop{\mathbb{E}}_{\theta,\theta'\sim q}[(rm(x,y,\theta)-\mathop{\mathbb{E}}_{x^*,y^*\sim p}[rm(x^*,y^*,\theta)])(rm(x,y,\theta')-\mathop{\mathbb{E}}_{x^*,y^*\sim p}[rm(x,y,\theta')])]]$$

(Swapping expectations)

$$\begin{aligned} &= \mathop{\mathbb{E}}_{\theta,\theta'\sim q}[\mathop{\mathbb{E}}_{x,y\sim p}[(rm(x,y,\theta)-\mathop{\mathbb{E}}_{x^*,y^*\sim p}[rm(x^*,y^*,\theta)])(rm(x,y,\theta')-\mathop{\mathbb{E}}_{x^*,y^*\sim p}[rm(x,y,\theta')])]] \\ &= \mathop{\mathbb{E}}_{\theta,\theta'\sim q}[\mathbb{C}ov_{x,y\sim p}[(rm(x,y,\theta),rm(x,y,\theta')] \\ &= \mathop{\mathbb{E}}_{\theta,\theta'\sim q}[\rho_{x,y\sim p}(rm(x,y,\theta),rm(x,y,\theta'))\sqrt{\mathbb{V}ar_{x,y\sim p}[rm(x,y,\theta)]}\sqrt{\mathbb{V}ar_{x,y\sim p}[rm(x,y,\theta')]}] \end{aligned}$$

(Using Jensen's inequality)
We define average correlation $\tilde{\rho}$ as:

$$\frac{\mathbb{E}_{\theta,\theta'\sim q}[\rho(\theta,\theta')\sqrt{Var[\theta]}\sqrt{Var[\theta']}]}{\mathbb{E}_{\theta,\theta'\sim q}[\sqrt{Var[\theta]}\sqrt{Var[\theta']}]}$$

and using this we get the variance of the margin as:

$$\mathbb{V}ar[margin] = \tilde{\rho}\mathop{\mathbb{E}}_{\theta\sim q}[\sqrt{Var[\theta]}^2 \leq \tilde{\rho}\mathop{\mathbb{E}}_{\theta\sim q}[Var[\theta]] \tag{45}$$

We get the expected value of the variance of a tree.

$$\begin{aligned} &\mathop{\mathbb{E}}_{\theta\sim q}[Var[\theta]] = \mathop{\mathbb{E}}_{\theta\sim q}[\mathbb{E}[rm_{x,y\sim p}(x,y,\theta)]^2]-\mathop{\mathbb{E}}_{\theta\sim q}[\mathop{\mathbb{E}}_{x,y\sim p}[rm(x,y,\theta)]]^2 \\ &\implies \mathop{\mathbb{E}}_{\theta\sim q}[Var[\theta]] = \mathop{\mathbb{E}}_{\theta\sim q}[\mathbb{E}[rm_{x,y\sim p}(x,y,\theta)]^2]-s^2 \end{aligned}$$

(rm is a binary variable, can take values 1/0, hence its expected value ≤ 1, and its square ≤ 1)

$$\implies \mathop{\mathbb{E}}_{\theta \sim q}[Var[\theta]] \leq 1 - s^2$$

$$Var[margin] \leq \tilde{\rho}(1 - s^2) \tag{46}$$

This helps in getting a bound on generalization error.

$$ge \leq \frac{\tilde{\rho}(1 - s^2)}{s^2} \tag{47}$$

This implies that a good forest should have low correlation among trees and high strength.

Growing Forests:
A tree sampling algorithm should ensure low correlation among the trees of the forest and a good average margin. But there are some issues:

1. If we overfit the training data, the max margin we get is 1, which is a single tree.
2. Good average margin means means trees close to max margin and implies high accuracy on training data.
3. But this can lead to high correlation among the trees. This happens because they are computing on the same training data.

The basic recipe for growing forests is:

1. Sample m data points. Adding more and more points improves strength but increases correlation.
2. Grow a decision tree.
3. Use a random subset of features for each split. Picking all features here will improve the strength but also increase correlation.

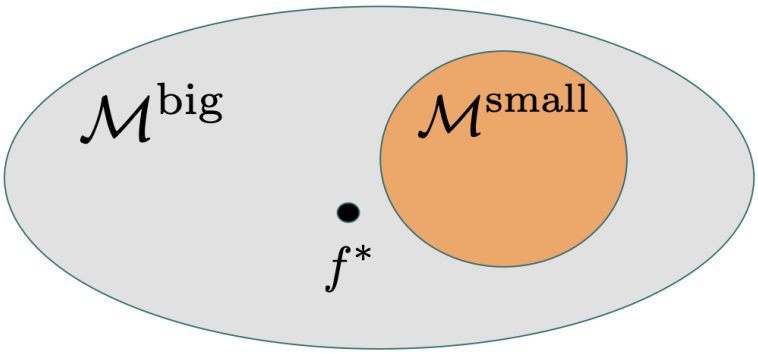


Figure 12: Model classes

We get a small model class when trees ignore the data. No matter the training data, we will get the same output. These trees are also independent. On the other hand, we get a big model class on using the training set completely. The model can adapt to the data set we have. The strength gives an idea of bias. Low strength shows high bias, it does not fit the data well.

- Strong trees $\implies$ big models
- Low correlation among trees $\implies$ small models

Important Points:

- *Multiple classes and regression*: We can draw a similar analysis for multiple classes. We can use mean squared error as a measure while using random forests for regression.
- *When to use Random Forests?*: When we have tabulated data, no rich data like signals, text or images, then random forests will give a good estimate.
- *Parallelism*: Each tree in a random forest requires a $\theta_k \sim q$, where every sample is drawn in parallel. This makes the random forests highly parallel.
- *Randomness of Random Forests*: Say there are n features, and for each split half the features are chosen, which determine the feature class. This gives a very rough estimate for building trees correctly, and it is exponential in n or $n/2$ (as at each level, we have $1/2$ chance of including the correct feature, and we need to hit $n/2$ features to get the answer). We can keep doing this repeatedly. This randomness helps prevent overfitting as each tree uses the data in a different way. Since everything is independent, the computation becomes faster too. Randomness also makes learning very fast. However, due to the randomness, it becomes difficult to find corners in the input space.

Moving from random search, we now shift our focus to functional gradients.

6.4 Functional Gradients

Here gradients, and not randomness, points towards the direction of interest. Gradients is done classically for vectors. Functions can be considered as infinite vectors. A n-dimensional vector is like a function mapping $0, \dots n-1$ to a number. Functions are like a big vector with many values to return.

Gradient Optimization for Functions:
The loss function for any function f is averaged across all the samples that we have. Let this be the expectation for now. It gives a notion of distribution between the label y and $f(x)$.

$$\mathscr{L}(f) = \mathop{\mathbb{E}}_{y,x} d(y, f(x)) = \mathop{\mathbb{E}}_{X} \mathop{\mathbb{E}}_{y|x} [d(y, f(x))] \tag{48}$$

Now we want to calculate the gradient at f^*. Let $f_{m-1}(x)$ be defined as:

$$f_{m-1}(x) = \sum_{i=1}^{m-1} \rho_i g_i(x) \tag{49}$$

The gradient will be found by finding the derivative of this.

$$g_m(x) = \frac{\partial \mathbb{E}_y[d(y, f(x))|x]}{\partial f(x)}\bigg|_{f=f_{m-1}} \tag{50}$$

We can interchange the differentiation and integral by pushing the derivative inside the expectation.

$$g_m(x) = \mathbb{E}_y\left[\frac{\partial d(y, f(x))}{\partial f(x)}|x\right]\bigg|_{f=f_{m-1}} \tag{51}$$

Once we get the gradient, we can update the function by subtracting the gradient from it. We minimize loss by following negative gradient. Let the step size of negative gradient be ρ_m. Updating the function with negative gradient looks like:

$$f_m(x) = \sum_{i=1}^{m-1} \rho_i g_i(x) - \rho_m g_m(x) \tag{52}$$

In some cases, computing the conditional expectation would give us the exact gradient. But we may not know what the conditional expectation is as we may have a single sample or no samples for many values of x. It is hard to compute gradient given only finite data. The few options we have for this case are:

1. Optimizing over a parametric family: We know the gradients at some values. These are the values for which we have the training data. We can try to make the function that we are learning close to the gradient of these values.

2. Using a greedy stage wise approach.

Greedy Function Optimization:
We can estimate h in 2 steps:

$$f_m(x) = \sum_{i=1}^{m-1} \rho_i h_i(x; a_i) + \rho_m h(x; a_m) \tag{53}$$

1. Get an estimate of the gradient and value x_i by evaluating $-g_m(x_i) = -\frac{\partial d(y_i, f(x_i))}{\partial f(x_i)}\Big|_{f=f_{m-1}}$
2. Project this gradient onto some family h that takes x_i as parameter.
   $min_a \sum_{i=1}^{N} [-g_m(x_i) - h(x_i; a_m)]^2$

We take ρ as the scaling factor that minimizes the error on the training data. We want to update f by the projected function. The functional gradient is projected onto some family. The following shows a kind of projected functional gradient, with the step size set by line search.

$$\rho_m = argmin_\rho \sum_{i=1}^{n} d(y_i, f_{m-1}(x_i) + \rho h(x_i; a_m)) \tag{54}$$

The prediction error can be taken as the mean squared error between the label and prediction:

$$d(y_i, f(x_i)) = \frac{1}{2}(y_i - f(x_i))^2 \tag{55}$$

The current predictor is taken as the sum of a bunch of trees and step size.

$$f_{m-1}(x) = \sum_{i=1}^{m-1} \rho_i h_i(x; a_i) \tag{56}$$

We compute the derivative at a known point f_{m-1}. Finally we will project the derivatives onto a tree.

$$-\frac{\partial d(y_i, f(x_i))}{\partial f(x_i)}\Big|_{f=f_{m-1}} = (y_i - f(x_i)) \tag{57}$$

This is the functional gradient. Essentially, we are building a better tree by locally judging the performance of the previous tree. We want a function that outputs this functional gradient. We project it by taking a new label and fitting a decision tree. We use the error of the current predictor as the real values using which we can construct the tree. To get a tree, we take the functional gradient at some points subject to some noise. We create a pseudo label $\hat{y}_i$ which is the functional gradient at the point that we know. We fit a tree on $(\hat{y}_i, x_i)$ pairs where $\hat{y}_i$ is:

$$\hat{y}_i = y_i - f(x_i) \tag{58}$$

Can we Train Forever?
At first, the error will keep getting smaller. The functional gradient will become 0 when $y_i = f(x_i)$. The number of trees will increase. But this will lead to the issue of over fitting. So we need to stop at some point. We can stop based on the number of iterations by checking the performance of the model against some validation set.

6.5 Random Forest vs Gradient Boosted Trees

1. *Overfitting:* Adding more trees in a random forest, given the fact that generalisation depends on the sampling distribution, does not lead to overfitting. The computational steps increase. Gradient boosted trees on the other hand can have issues of overfitting and more computational steps on adding trees.
2. *Parallelism:* In random forest, all trees can be trained in parallel as all trees have their own parameter θ_k, all from from the same distribution. Gradient boosted trees undergo sequential training and so for very large data sets, it might take more time. We can improve this by extreme gradient boosting.
3. *Selecting the input subspace:* In random forests, we have a random predictor randomly selecting subset of features for tree splitting. We get the right input regions randomly. In gradient boosted trees, the goal is to minimize the local error by computing residuals and fitting them. We get the right input regions through gradients.

7 3/8/2022 Latent Variable Models

We've seen models that describe a relationship between some features, $\mathbf{x}$, and a label y, both of which are observed. Sometimes, we wish to model things which aren't observed, or *latent* variables. Latent variables may correspond to real things we'd like to make inferences about, or may be imaginary components of our model that simplify it or describe important relationships in the data. Early applications of latent variables, for example, focused on modeling the effects of 'general intelligence,' which is an abstract concept that is impossible to actually measure, but that can be approximated using scores from different tests of cognitive performance (e.g., memory, verbal, spatial, etc.).

7.1 Recap of Previous Topics

We've covered:

- Generalized linear models for predicting many kinds of output data types,
- (Stochastic) gradient descent for optimization,
- Regularization to improve generalization and control model complexity, especially when there are more features than data points,
- Understanding generalization error in terms of the bias-variance, tradeoff (too much bias leads to underfitting, too much variance leads to overfitting),
- Neural networks and reducing variance via weight sparsity and parameter sharing (e.g., convolutional neural networks),
- Random forests and how adding more trees to the ensemble doesn't cause overfitting,
- The difficulty of finding specific feature combinations with random forests,

7.2 Breakout Question 1

> "A model trained with finite data perfectly fits the training data, but does well on test. Is this possible?"

Model classes with high capacity, e.g. deep neural networks, contain multiple solutions that can fit the training data well, but which perform differently on test data. How does the learning algorithm "choose" between these equally good functions? The algorithm must have some inductive biases. All algorithms, due to the same industrial biases, produce different decision boundaries. We can add prior information to constrain the model exploration. Other factors, including things like the optimization algorithm or initialization method, plays a role in deciding which solution gets chosen.

7.3 Generative Modeling

A generative model includes the distribution of the data itself, and tells you how likely a given example is. For example, models that predict the next word in a sequence are typically generative models (usually much simpler than GANs) because they can assign a probability to a sequence of words. A discriminative model ignores the question of whether a given instance is likely, and just tells you how likely a label is to apply to the instance. A *discriminative* model learns a conditional distribution for the labels given some the features: $p_\theta(y|\mathbf{x})$. A *generative* model learns a joint distribution for the features and the labels: $p_\theta(\mathbf{x}, y)$.

Motivation for generative models One example is if some $x_{i,j}$ are missing, and we need to model both the target y as well as $x_{missing}$. Since some $x_{i,j}$ are missing, it is necessary to model x, as we need to predict the missing features, which can be done using the joint distribution and the marginals:

$$p(\mathbf{x}_{\text{missing}}|\mathbf{x}_{\text{observed}}) = \frac{p(\mathbf{x}_{\text{missing}}, \mathbf{x}_{\text{observed}})}{p(\mathbf{x}_{\text{observed}})} = \frac{\int p(y, \mathbf{x}_{\text{missing}}, \mathbf{x}_{\text{observed}})dy}{\int p(y, \mathbf{a}, \mathbf{x}_{\text{observed}})dy\,d\mathbf{a}} \tag{59}$$

Once the missing features are filled out, we can make predictions using

$$p(y|\mathbf{x}) = \frac{p(y, \mathbf{x})}{p(\mathbf{x})} = \frac{p(y, \mathbf{x})}{\int p(y, \mathbf{x})dy} \tag{60}$$

Conceptually, there is no difference between **x** and y in p(**x**,y); they are just variables. When we predict "y", it's just a missing variable. However, predicting with missing data features is more complex, and at the very least requires a model of the data itself.

7.4 Probabilistic Latent Variable Models

We can use *latent* (hidden) variables **z** to represent unobserved structures in the model that we might be interested in learning. For example, in medicine, we may "observe" patients in different hospital sections like "Geriatrics" or "Maternity", but what is hidden are the qualities that these patients share, such as abdominal pain or pregnant state. Even though latent variables are not observed, they affect the data generating process and impact data which we do observe.

Given observed **x** and latent **z**, we now want to model the joint distribution p(**x**,**z**) in terms of the prior p(**z**) and the likelihood p(**x**|**z**):

$$p(\mathbf{x}, \mathbf{z}) = p(\mathbf{z})p(\mathbf{x}|\mathbf{z}) \tag{61}$$

Here, the prior represents the belief we have for the latent variable, and represents some real-world knowledge we have about the observation. The likelihood defines for each hidden structure, how likely the observation is.

Example: Mixture of Gaussians Assume that the data is generated from a mixture of two Gaussians:

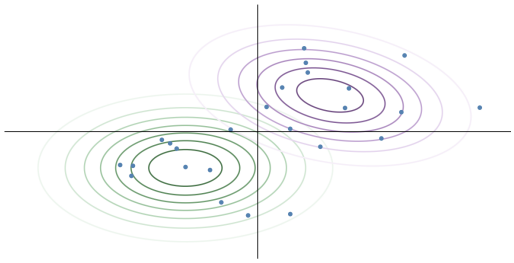


Figure 13

The hidden variable $\mathbf{z}_i$ is a categorical variable, indicating which Gaussian generated the data point $\mathbf{x}_i$. In the picture, the hidden variable is the class or color that the data point should have.

Modeling Define a *prior* over the hidden variable. For example, assign each class $\frac{1}{2}$ probability:

$$\mathbf{z}_i \sim \mathrm{C}ategorical(\frac{1}{2}, \frac{1}{2})$$

Next, define a *likelihood* specifying how to generate a point in each class:

$$\mathbf{x}|\mathbf{z}_i = k \sim \mathcal{N}(\mu_k, \Sigma_k)$$

where μ_k and Σ_k are the means and covariances of the two Gaussian components. The parameters for our model are then $\theta = \{\mu_k, \Sigma_k\}_{k=1}^K$, where $K = 2$.

The joint or total likelihood over both **x** and **z** is

$$p_\theta(\{\mathbf{x}_i, \mathbf{z}_i\}_{i=1}^N) = \prod_{i=1}^N p(\mathbf{z}_i)p(\mathbf{x}_i|\mathbf{z}_i) \tag{62}$$

The joint likelihood captures our assumption that each of the N data points were drawn independently.

Given an observed data point, $\mathbf{x}_i$, we can predict the class it came from using the *posterior* over the hidden variable:

$$p_\theta(\mathbf{z}_i|\mathbf{x}_i) = \frac{p(\mathbf{z}_i)p(\mathbf{x}_i|\mathbf{z}_i)}{p(\mathbf{x}_i)} \tag{63}$$

Example :

$$P(Z=1|X) = \frac{\frac{1}{2}\mathcal{N}\left(\mu_1, \sum_1\right)}{\frac{1}{2}\mathcal{N}\left(\mu_1, \sum_1\right) + \frac{1}{2}\mathcal{N}\left(\mu_2, \sum_2\right)}$$

$$P(Z=k|X) = \frac{\frac{1}{K}\mathcal{N}\left(\mu_k, \sum_k\right)}{\sum_{i=i}^{i=K} \frac{1}{k}\mathcal{N}\left(\mu_i, \sum_i\right)}$$

Learning We'd like to apply maximum likelihood estimation; however, we only have access to $\mathbf{x}$, not $\mathbf{z}$. Thus, it's not possible to optimize $p(\{\mathbf{x}_i, \mathbf{z}_i\}_{i=1}^N)$. Instead, we need to marginalize out $\mathbf{z}$:

$$\log p_\theta(\{\mathbf{x}_i\}_{i=1}^N) = \sum_{i=1}^N \log \sum_{k=1}^K p(\mathbf{z}_i = k) p(\mathbf{x}_i|\mathbf{z}_i = k; \mu_k, \Sigma_k) \tag{64}$$

We will learn $\theta = \{\mu_k, \Sigma_k\}_{k=1}^K$ by optimizing (64), e.g. via taking gradients. This is *maximum likelihood estimation*. It is equivalent to minimizing the KL-divergence $KL\left(p_\theta(\{\mathbf{x}_i\}_{i=1}^N) || p_{\mathscr{D}}\right)$ where $p_{\mathscr{D}}$ is the (true) data-generating distribution. Therefore, maximum likelihood estimation moves the model closer to the data generating distribution.

To see this equivalence more clearly, consider model p_θ and data distribution F. If we maximize the likelihood, the objective is the following:

$$\max_\theta \mathbb{E}_F[\log p_\theta] \tag{65}$$

This objective does not change if we subtract a constant:

$$\max_\theta \mathbb{E}_F[\log p_\theta] = \max_\theta \mathbb{E}_F[\log p_\theta] - \mathbb{E}_F[\log F] \tag{66}$$

Now, with a bit of manipulation, we see that the term to be maximized is the same as negative KL-divergence, and thus maximizing the likelihood is the same as minimizing KL-divergence:

$$\mathbb{E}_F[\log p_\theta] - \mathbb{E}_F[\log F] = \mathbb{E}_F[\log p_\theta - \log F] = \mathbb{E}_F[\log \frac{p_\theta}{F}] = -\mathbb{E}_F[\log \frac{F}{p_\theta}] = -\mathrm{KL}(p_\theta||F) \tag{67}$$

Takeaway: **Maximum Likelihood is equivalent to minimizing the KL-Divergence between the model and the data generating distribution.**

Learning Meaningful Latent Variable Models What would have happened if we'd used more flexible distribution families instead of the Gaussian? The model could have made each mixture component equivalent to the data generating process. Then, the latent classes, $\mathbf{z}_i$, would be meaningless and the latent variable would capture no interesting structure.On the other hand, if we use a more rigid distribution or don't consider the data generating process, then we have less function to explore while optimizing. Thus, the solution would be constrained and not optimal.

In order to learn meaningful structure, latent variable models have to make assumptions about how the data was generated.

7.5 Graphical Models

Most probabilistic paradigms can be solved via the sum rule and the product rule, but an equally potent analysis uses diagrammatic representations of probability distributions known as probabilistic graphical models- a graph where nodes (open circles) are random variables (smaller, solid points represent deterministic parameters) and the edges are the relationship between them. To highlight the difference between latent and observed variables, the nodes representing observed random variables may be shaded in.

So we can say that A graphical model or probabilistic graphical model (PGM) or structured probabilistic model is a probabilistic model for which a graph expresses the conditional dependence structure between random variables.Generally, probabilistic graphical models use a graph-based representation as the foundation for encoding a distribution over a multi-dimensional space and a graph that is a compact or factorized representation of a set of independences that hold in the specific distribution. Two branches of graphical representations of distributions are commonly used, namely, Bayesian networks and Markov random fields. Both families encompass the properties of factorization and independences, but they differ in the set of independences they can encode and the factorization of the distribution that they induce

A *Bayesian network* is a kind of directed graphical model. It is a probabilistic graphical model for representing knowledge about an uncertain domain where each node corresponds to a random variable and each edge represents the conditional probability for the corresponding random variables. It provides a way to describe how the variables of the data-generating process are related. In this setting, we express a probabilistic model $p(\mathbf{x}_{[1]}, \ldots, \mathbf{x}_{[M]})$ (here $\mathbf{x}_{[i]}$ indicates the i-th feature, not the i-th data point) using a directed acyclic graph (DAG) representation, with the M nodes being random variables $\mathbf{x}_{[m]}$ and the (directed) edges being relationships between variables. Directed graphs are constructed by introducing a node for each random variable, and for each conditional distribution we add directed links as follows. If we want to represent the conditional probability $p(c|b, a)$ we add directed links from a to c (here, a is a parent of c ($\text{pa}_c = \{\text{a}\}$) and from b to c. A graph G specifies the joint distribution of $\mathbf{x}_{[1]}, \ldots, \mathbf{x}_{[M]}$ as a product over all graph nodes of the conditional distribution of each node, conditioned on its parents:

$$p(\mathbf{x}_{[1]}, \ldots, \mathbf{x}_{[M]}) = \prod_{m=1}^{M} p(\mathbf{x}_{[m]}|pa_m) \tag{68}$$

where pa_m is the set of all parent nodes of $\mathbf{x}_{[m]}$ in G. For example, the graphical model below

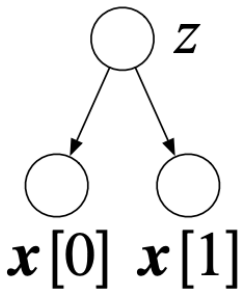


Figure 14

can be factorized as $p(\mathbf{z}, \mathbf{x}_{[0]}, \mathbf{x}_{[1]}) = p(\mathbf{z})p(\mathbf{x}_{[0]}|\mathbf{z})p(\mathbf{x}_{[1]}|\mathbf{z})$. In this way, graphical models are a powerful way of identifying conditional dependencies and independencies. To test whether variables are conditionally dependent or independent, we aim to derive the independence condition based on the joint distribution observed from the graph. For example, say we want to test whether $\mathbf{x}_{[0]}$ is conditionally independent of $\mathbf{x}_{[1]}$ given $\mathbf{z}$. To check, we use the following reasoning:

$$p(\mathbf{x}_{[0]}, \mathbf{x}_{[1]}|z) = \frac{p(\mathbf{z}, \mathbf{x}_{[0]}, \mathbf{x}_{[1]})}{p(\mathbf{z})} = \frac{p(\mathbf{z})p(\mathbf{x}_{[0]}|\mathbf{z})p(\mathbf{x}_{[1]}|\mathbf{z})}{p(\mathbf{z})} = p(\mathbf{x}_{[0]}|\mathbf{z})p(\mathbf{x}_{[1]}|\mathbf{z})$$

independent from each other.

To summarize, graphical models allow us to visually represent the joint probability distribution over all the random variables in a model. Conditional dependence is represented by the presence of an edge between the parent and child node. Conditional independence is represented by the lack of an edge between two nodes. The conditional independence assumptions introduced in the graphical model allow us to simplify the joint distribution product rule into a more compact form.

A notational convenience is plate notation, shown in the above graphical view. Plate notation may be used to denote a collection of variables that consist of the same "kind" of variable that repeats in the

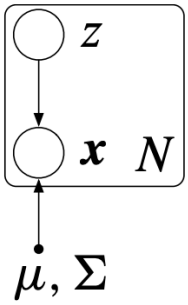


Figure 15

model a certain number of times. The above graph shows that there are N variables from the same distribution as $\mathbf{x}$, and N variables from the same distribution as z in the model.

7.6 Evaluating Latent Variable Models

Just like discriminative models, we can evaluate latent variable models in the same way, which is by evaluating $\log p_\theta(\mathbf{x})$ on a held-out validation set $\{\mathbf{x}\}_{i=1}^{M}$.

7.7 Breakout Question 2

> What would be a good latent variable model where there's an underlying state that's responsible for dependence in a sequence of noisy measurements?

We can use a latent variable, $\mathbf{z}_t$ to model the state at time t. Since the state is responsible for the dependence, the observations ($\mathbf{x}_t$) should be independent conditioned on the state. Finally, the states must depend on each other in order to model the dependence. In the simplest case, $\mathbf{x}_t$ could depend solely on $\mathbf{z}_t$, and $\mathbf{z}_t$ could depend solely on $\mathbf{z}_{t-1}$. This is called a *Hidden Markov model*.

7.8 Topic Modeling

Setup We have N documents, each of which contains V words. The observed data is therefore the $N \times V$ matrix of words W, where the order of documents is not relevant. The hidden structure here is the **topic**. A topic t_k is a distribution β_k over words. Furthermore, each document d_i is generated from a distribution θ_i over topics. So $\{\beta_k, \theta_i\}$ are the parameters of interest.

Now, we want to construct our model. The full generative model is: We can use Dirichlet priors over β_k and θ_i. That is, each topic has distribution $\beta_k \sim Dirichlet(\alpha)$ and each document has distribution $\theta_i \sim Dirichlet(\kappa)$. The likelihood is then Categorical. Hence, for word m in document l, we have the word's topic $z_{m,l} \sim Categorical(\theta_i)$ and the word from $w_{m,l} \sim Categorical(\beta_{z_{m,l}})$.

Inference The quantity that we want to infer is the **posterior** $p(\beta, \theta, \mathbf{z}|W)$. This is, in general, difficult.

8 3/22/2022 Computing with Latent Variable Models

8.1 Q/A

Recall our setup from last time:

- Data: x
- Latent Variables: z
- Model: $p(x,z) = p(x|z)p(z)$
- Posterior: $p(z|x)$ - probability of hidden structure

Recall a motivating example: Lets say we want to identify the light sources in an image from space. We can use a Latent Variable (LV) Model, treating each source of light as an LV. Note that we have some knowledge of light measures from physics:

$p(x_i|z_i = galaxy)$ where x_i = light measurement

Want to compute $p(z_i = galaxy|x_i)$:

$$p(z_i = galaxy|x_i) = \frac{p(z_i = galaxy)p(x_i|z_i = galaxy)}{\sum p(x_i|z_i = type)p(z_i = type)}$$

Uses of Latent Variable models:

- Encode prior knowledge
- Combine simple distribution to create more complex ones
- Uncovering hidden structure

Combining Distributions: Take a categorical distribution Categorical(1...K) and a normal distribution Normal(,); combine to get a mixture of Gaussians, which is far more flexible:

$$z_i \sim Categorical(1...K)$$
$$x_i \sim Normal(_i,_i)$$

Questions from Last Year:

- Q: Do LVs help predict X?
  A: Sort of
- Q: How to know if LV is correct? Can I use cross-validation like Y?
  A: Cannot be checked without assumptions b/c data gen. distribution $F(x)$ is all you get from the data. $F(x)$ can be modeled w/out latent variables. For a fixed class, predictions might help
- Q: Are latent variables about the noise and getting an accurate data generating distribution?
  A: Latent variables can be used as noise variables, can exist in concert w/ noise variables

8.2 Example: find topics in documents

Setup:

- Data: M documents, V words, W: M x V matrix of words (wordcount matrix)
- Hidden Structure: Group of topics that describe the document. Each document contains a distribution over topics:
  - β_i - a distribution over words for each topic
  - θ_i - a distribution over topics for each document

  Topic - distr. of words

Priors? Dirichlet distribution (needed for joint distribution)

Algorithm: For each word m in document l:

1. For each topic, draw distribution over words from Dirichlet(α)
2. For each document, draw distribution over topics from Dirichlet(κ)
3. For word m in document l:
   - Draw word's topic from $Z_m \sim Categorical(\theta_i)$
   - Draw word from $W_{m,l}\ Categorical(\beta_{z_{m,l}})$
4. Compute posterior:

$$p(\beta, \theta, z|w) = \frac{p(\beta, \theta, z, w)}{p(w)}$$

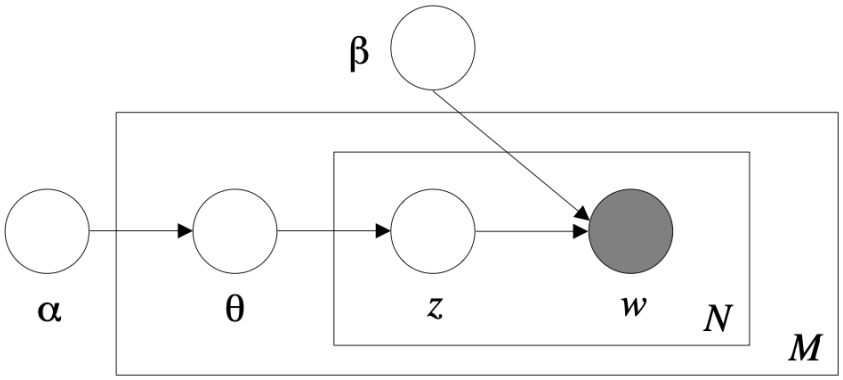


Figure 16: Graphical model representation of topic model

Computing the posterior, is it easy? No. Computing the high dimensional integral in the marginal generally analytically intractable challenging:

$$p(x) = \int p(x, z)\, dz$$

How about for a Bayesian Mixture of Gaussians?

$$\mu_k \sim Normal(0, 1)$$
$$z_i \sim Categorical(1...k)$$
$$x_i \sim Normal(\mu_{zi}, 1)$$

Marginal likelihood is:

$$p(x) = \int \prod_{k=1}^{K} p(\mu_k) \prod_{i=1}^{N} \sum_{j=1}^{K} p(z_i = j) p(x_i|\mu_j) d\mu_1 ... d\mu_k$$

Swapping:

$$p(x) = \sum_{z} \int \prod_{k=1}^{K} p(\mu_k) \prod_{i=1}^{N} p(z_i = j) p(x_i|\mu_j) d\mu_k$$

$$p(x) =$$

8.3 Variational inference (VI)

Recall that we want to compute the posterior $p(z|x)$. Posit a family of distributions $q(z;\lambda)$ w/ parameter λ. Find $q(z) = p(z|X = D)$.

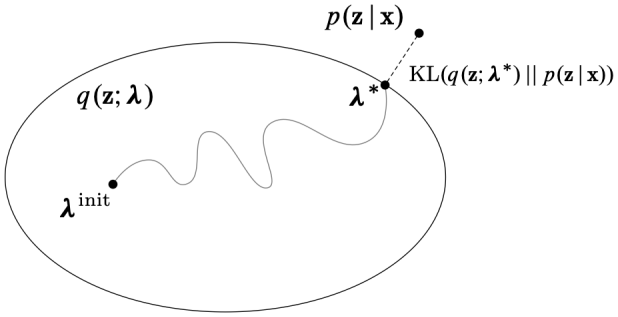


Figure 17

Minimize $KL\ (q(z;\lambda^*)||p(z|x))$. While we don't have $p(z|x)$, we can refactor as follows:

$$\begin{aligned} KL\ (q(z;\lambda^*)\ ||\ p(z|x)) &= \mathbb{E}_q[\ logq(z,\lambda) - log(p(z|x)\ ] \\ &= -\mathbb{E}_q[logq(z,\lambda) - logp(z,x) + logp(x)] \\ &= -\mathbb{E}_q[logq(z,\lambda) - logp(z,x)] + logp(x)] \end{aligned}$$

Key point: notice that maximizing the ELBO term here minimizes the KL divergence:

$$logp(x) = KL + \mathscr{L}$$

where KL is a positive term, $\mathscr{L}$ is a negative term, and $logp(x)$ is a negative term.

$$\mathscr{L}(\lambda) = \mathbb{E}_q[logp(x,z)] - \mathbb{E}_q[logq(z;\lambda)]$$

The lower bound on $log(p(x))$:

$$\begin{aligned} logp(x) &= \mathbb{E}_q[logp(x,z)] - logq(z;\lambda)] + KL((q(z;\lambda)||p(z|x)) \\ &\geq \mathbb{E}_q[logp(x,z)] - logq(z;\lambda)] \end{aligned}$$

ELBO trades off two terms. q is chosen to match types

- The first term prefers q(·) to place its mass on the MAP estimate
- The second term encourages q(·) to be diffuse

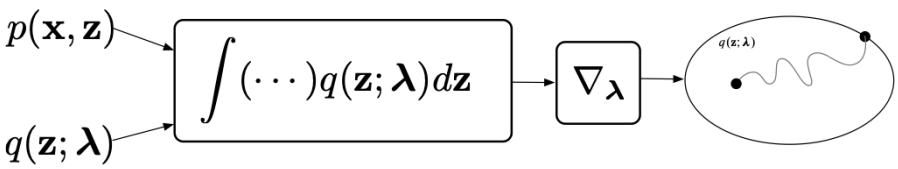


Figure 18: The recipe

VI recipe:

1. Start w/ p(z,x)
2. Choose variational approx $q(z;\lambda)$
3. Write down ELBO: $\mathscr{L}(\lambda) = \mathbb{E}_q[logp(x,z)] - \mathbb{E}_q[logq(z;\lambda)]$
4. Compute expectation: $\mathscr{L}(\lambda) = x\lambda^2 + log\lambda$
5. $\nabla\mathscr{L}(\lambda) = 2x\lambda + \frac{1}{\lambda}$
6. Maximize using gradient ascent: $\lambda_{t+1} = \lambda_t + p_t\nabla_\lambda\mathscr{L}$

Example: Bayesian Linear Regression

$$p(w) \sim N(0,1)$$

$$p(y_i|x_i,w) \sim Bernoulli(\sigma(wx_i))$$

Assume one point (y, x). The approximating family q is the normal; $\lambda = \mu, \sigma^2$

Evaluating the ELBO:

$$\mathscr{L}(\mu,\sigma^2) = \mathbb{E}_q[logp(z)] - logq(z) + logp(y|x,z)] \tag{69}$$

$$= -\frac{1}{2}(\mu^2+\sigma^2) + \frac{1}{2}log\sigma^2 + yx\mu - \mathbb{E}_q[log(1+exp(xz))] \tag{70}$$

Stuck. Cant evaluate the last term (i.e cannot take the expectation). And the expectation hides the objectives dependence on the variational parameters. This makes it hard to directly optimize. We could derive model specific bounds, but this wouldn't generalize and would require tedious computation for different types of models.

Idea: use modified version of VI recipe where taking the gradient and the integral are switched ... use stochastic optimization.

Computing the gradient of an expectation. First, define:

$$g(z,\lambda) = logp(x,z) - logq(z;\lambda)$$

Compute $\nabla_\lambda\mathscr{L}$:

$$\nabla_\lambda\mathscr{L} = \nabla_\lambda \int q(z;\lambda)g(z,\lambda)dz \tag{71}$$

$$= \int \nabla_\lambda q(z;\lambda)g(z,\lambda) + q(z;\lambda)\nabla_\lambda g(z,\lambda)dz \tag{72}$$

$$= \int q(z;\lambda)\nabla_\lambda logq(z;\lambda)g(z,\lambda) + q(z;\lambda)\nabla_\lambda g(z,\lambda)dz \tag{73}$$

$$= \mathbb{E}_{q(z,\lambda)}[\nabla_\lambda logq(z;\lambda)g(z,\lambda) + \nabla_\lambda g(z,\lambda)] \tag{74}$$

Thus, we can write the ELBO gradient as an expected value. This will let us compute the score function gradient and reparameterization gradients which will help us optimize ELBO.

8.4 score function gradients

Recall that if we can move the gradient of ELBO inside the intergration, we have

$$\nabla_\lambda\mathscr{L} = \mathbb{E}_{q(z,\lambda)}[(\nabla_\lambda logq(z;\lambda)g(z,\lambda)) + \nabla_\lambda g(z,\lambda)] \tag{75}$$

Note that (The calculation is left as after class exercise, I am not certain about the correctness of the calculation, feel free to point out my mistakes)

$$\begin{aligned}
\mathbb{E}_{q(z,\lambda)}[\nabla_\lambda g(z,\lambda)] &= \mathbb{E}_{q(z,\lambda)}[\nabla_\lambda(\log p(x,z) - \log q(z;\lambda))] \\
&= \mathbb{E}_{q(z,\lambda)}[\nabla_\lambda(\log p(x,z)) - \nabla_\lambda(\log q(z;\lambda))] \\
&= \mathbb{E}_{q(z,\lambda)}[-\nabla_\lambda(\log q(z;\lambda))] \\
&= -\mathbb{E}_{q(z,\lambda)}[\frac{\nabla_\lambda(q(z;\lambda))}{q(z;\lambda)}] \\
&= -\int q(z,\lambda)\frac{\nabla_\lambda(q(z;\lambda))}{q(z;\lambda)}dz \\
&= -\nabla_\lambda \int q(z;\lambda)dz \\
&= -\nabla 1 \\
&= 0
\end{aligned}$$

So

$$\nabla_\lambda \mathscr{L} = \mathbb{E}_{q(z,\lambda)}[(\nabla_\lambda \log q(z;\lambda))g(z,\lambda) + \nabla_\lambda g(z,\lambda)] \tag{76}$$

$$= \mathbb{E}_{q(z,\lambda)}[(\nabla_\lambda \log q(z;\lambda))(\log p(x,z) - \log q(z;\lambda))] \tag{77}$$

In the real life, we will draw S data from the distribution, so we can compute this noisy unbiased gradient with Monte Carlo by

$$\frac{1}{S}\sum_{s=1}^{S}(\nabla_\lambda log q(z_s;\lambda))(log p(x,z_s) - log q(z_s;\lambda))$$

and we use this gradient to update the value of λ similar to the way we did gradient descent.

In this process, we need data drawn from $q(z)$, $\nabla_\lambda \log q(z_s;\lambda)$, and $\log p(x,z_s)$ and $\log q(z_s;\lambda)$. There is no model specific work, so this satisfies black box variational inference criteria.

However, score function gradient inference has one problem, which is the large gradient variance.

$$Var_{q(z;v)} = \mathbb{E}_{q(z;v)}[(\nabla_v \log q(z;v)(\log p(x,z) - \log q(z;v)) - \nabla_v \mathscr{L})^2] \tag{78}$$

In other words, if our dataset contains a lot of rare data, the gradients we use to update λ will also have high variance. If we want to lower the variance, in each step of updating λ, we need to average large amount data to compute the gradient. If we keep the amount of data participating each step of updating λ low, we have to make only small steps to prevent high variance caused by rare data. In both cases, the computational costs are high.

One solution to solve this problem is control variates, namely, we find another function $\hat{f}$, where $E[\hat{f}(z)] = E[f(z)]$ but $Var(\hat{f}(z)) < Var(f(z))$, to replace f itself. The formula is

$$\hat{f}(z) := f(z) - a(h(z) - \mathbb{E}[h(z)]) \tag{79}$$

where we (had better) choose h having high correlation with f and choose a to minimize the variance. In variational inference, a good choice for h is $\nabla_\lambda q(z;\lambda)$ partly because $\mathbb{E}_q[\nabla_\lambda q(z;\lambda)] = 0$ for any q.

8.5 reparametrization gradients

In score function gradient estimation, we sample from $q(z)$, evaluate $\nabla_\lambda \log q(z;\lambda)$, $\log p(x,z)$, and $\log q(z)$. However, we might evaluate a different kind of gradients if we make less strict assumption. In this case, we further assume that

1. $z = t(\epsilon, \lambda)$, that is z is a function of noise ϵ and parameters λ. One nice property about ϵ is that it is independent from λ. In other words, we have a distribution $s(\epsilon)$, which implies $z \sim q(z; \lambda)$
   For example, we can choose $\epsilon \sim N(0, 1)$ and $z = \epsilon\sigma + \mu$. This implies that $z \sim q(z; \lambda) = N(\mu, \sigma^2)$.
2. $\log p(x, z)$ and $\log q(z)$ are differentiable with respect to z.

If these assumptions satisfy, we can then invent a new kind of gradient, reparametrization gradients, or pathwise gradients.

The calculation comes as following:

1. Recall from previous sections
$$\nabla_\lambda \mathscr{L} = \mathbb{E}_{q(z,\lambda)}[(\nabla_\lambda log q(z; \lambda) g(z, \lambda)) + \nabla_\lambda g(z, \lambda)]$$
2. Rewrite it using $z = t(\epsilon, \lambda)$ using properties that we can separate $s(\epsilon)$ from $q(z; \lambda)$:
$$\nabla_\lambda \mathscr{L} = \mathbb{E}_{s(\epsilon)}[(\nabla_\lambda log s(\epsilon) g(t(\epsilon, \lambda), \lambda)) + \nabla_\lambda g(t(\epsilon, \lambda), \lambda)] \tag{80}$$
3. Note that $\log s(\epsilon)$ contains no λ so $\nabla_\lambda log s(\epsilon) = 0$, so the expression can be further simplified:
$$\nabla_\lambda \mathscr{L} = \mathbb{E}_{s(\epsilon)}[\nabla_\lambda g(t(\epsilon, \lambda), \lambda)] \tag{81}$$
4. the following computations are:
$$\begin{aligned}\nabla_\lambda \mathscr{L} &= \mathbb{E}_{s(\epsilon)}[\nabla_\lambda g(z, \lambda)] \\ &= \mathbb{E}_{s(\epsilon)}[\nabla_z (\log p(x, z) - \log q(z; \lambda)) \cdot \nabla_\lambda z - \nabla_\lambda (\log q(z; \lambda))] \\ &= \mathbb{E}_{s(\epsilon)}[\nabla_z (\log p(x, z) - \log q(z; \lambda)) \cdot \nabla_\lambda t(\epsilon, \lambda)]\end{aligned}$$

This gives us pathwise gradient.

8.6 further discussion

Question1: What's an example problem that might have gradients where one is better than the other? Roughly speaking, the variance of one value will become really large when that value has been multiplied with a large constant, so if the gradient of function cannot keep constant within (abstractly, linear function), then score function gradient will work better.

comparison between score function estimator and reparametrization estimator Generally speaking, score function estimator differentiates the density ($\nabla_\lambda q(z; \lambda)$), so it works for a wide range of models, but it suffers from great variance of the rare data.

On the other hand, reparametriation function works with differentiation of the function ($\nabla_z (\log p(x, z) - \log q(z; \lambda))$), so we have to make sure the function is differentiable, but we do not suffer from high variance that much

How do we use both estimators at the same time? We can use the trick of $1 - \alpha$ and α to take the weighted average of two gradients. If the model specifies some specific requirements such as low variance, we can choose α to meet requirements.

Parameter Learning with Variational Inference The most original idea is to train model $p_\theta(x, z)$ by MLE, but the difficulty of integral makes this task difficult, and unknown prior prevents from deriving posterior, which blocks our way to avoid hard integral problem.

So we optimize the lower bound of likelihood instead, that is $\log p_\theta(x) \geq E_q[\log p(z, x) - \log q(z; \lambda)] := \mathscr{L}(\lambda, \theta)$.

Note that $\mathscr{L}(\lambda, \theta)$ might be a function of 2 variables. We can apply coordinate ascent (fixing

one variable and optimize another, then use the optimized value to optimize previously fixed variable).

If we fix the parameters and compute the optimal posterior $q_t = p_{\theta_{t-1}}(z|x)$ and optimize parameters with this posterior, then the algorithm is called Expectation Maximization algorithm. (EM)

9 3/29/2022 Deep Generative Models

9.1

The model $\int p(x,z)dz = p(x)$ can be used to find hidden structure.
Question - Why cannot sample from $p(x)$ directly?

9.2 Variational Autoencoder (VAE)

$$p(z) = \mathcal{N}(0,1)$$
$$p(x|z) = \mathcal{N}(\mu_\beta(z), \sigma^2_\beta(z))$$
$$\sum_{i=1}^{n} log p_\beta(x_i, z_i)dz_i$$

is intractable. Hence we should use variation inference instead.

9.2.1 Evidence Lower Bound (ELBO)

For some point, we would like to have

$$max_\beta log p_\beta(x) = \max_\beta log \int p_\beta(x,z)dz$$

Hence, the integral is intractable:https://www.overleaf.com/project/6209a6e4527075728c4b56b0

$$\begin{aligned}
log p_\beta(x) &= \mathbb{E}_q(log p_\beta(x)) \\
&= \mathbb{E}_q(log p_\beta(x,z) - log p_\beta(z|x)) \\
&= \mathbb{E}_q(log p_\beta(x,z) + log q(z;\lambda) - log q(z;\lambda) - log p_\beta(z|x)) \\
&= \mathbb{E}_q(log p_\beta(x,z) + log q(z;\lambda) + \mathbb{E}(log q(z;\lambda) - log p(z|x)) \\
&= \mathbb{E}_q(log p_\beta(x,z) + log q(z;\lambda) + KL(q(z;\lambda)||p(z|x)) \\
&\leq \mathbb{E}_q(log p_\beta(x,z) + log q(z;\lambda)
\end{aligned}$$

Hence, we can optimize lower bound on $log p$ w.r.t. λ and β.

Variational approximation:

$$q(z) = \Pi_{i=1}^n q(z_i; m_i, s_i)$$

Note that $q(z_i; m_i, s_i) \sim \mathcal{N}(m_i, s_i)$ and each data point has separate variational parameters.

$$\mathcal{L}(q,\beta) = E_q(log p - log q) = \sum_{i=1}^{n} \mathbb{E}(log p(z_i, x_i) - log q(z_i; m_i, s_i))$$

We can optimize this using reparameterization gradients.

9.3 Direct Likelihood Models

Using maximum likelihood to learn a model p_θ given samples $x_1, x_2, ..., x_n$: 1. Write down a model $p_\theta(x)$; 2. Write the observed likelihood of the data $L = \sum_{i=1}^n \log p_\theta(x_i)$; 3. Take gradient with respect to θ; 4. Use stochastic gradients to maximize the likelyhood of the data and the model should place high probability on the data.

Maximum Likelihood is good in that, define

$$\hat{F}(x) = \frac{1}{n}\sum_{i=1}^{n} \delta_{x_i}(x),$$

The maximum likelihood L is

$$L = \sum_{i=1}^{n} \theta(x_i) = nE_{\hat{F}}[\log p_\theta(x_i)] = nE_{\hat{F}}[\log p_\theta(x_i) - \log \hat{F}(x_i)] + C = -nKL(\hat{F}(x)||p_\theta x) + C$$

This shows that maximum likelihood minimizes the KL-divergence between the empirical distribution on the data to the model.

How to use maximum likelihood to fit observed data?

Idea 1: Decompose $p(x)$ using the chain rule, e.g. $p(x) = p(x_1)p(x_2|x_1)p(x_3|x_1, x_2)$. But as the dependency set grows, the conditional distribution parameter size grows exponentially. We can also use recurrent neural networks, which are sequence models $h_i = f_\theta(h_{i-1}, x_{i-1})$. The hidden state h_t has information about all the previous $x_{<i}$. Use this to build a probability model over x:

$$h_i = f_\theta(h_{i-1}, x_{i-1}),$$

$$p_\theta(x_i|x_{<i}) = p(x_i|h_i).$$

Idea 2: Transform Noise Vectors. Directly get k dimensional densities by transforming noise ϵ distributed by $s(\epsilon)$, $x = f_\theta(\epsilon)$. Assume f is invertible, then

$$p(x) = s(f^{-1}_{theta}(x))|\frac{df^{-1}_{theta}(x)}{dx}|$$

But this $p(x)$ requires computing the determinant of a $k \times k$ matrix. The determinant computation in general is cubic, so sampling and density requires both f and its inverse to be easy to compute. The solution is to build a function with lower triangular Jacobian since the determinant of lower triangular matrix can be computed in $O(n)$. To do so we can limit the dependencies between dimensions such as

$$x_1 = f_\theta(\epsilon_1)$$

$$x_2 = f_\theta(\epsilon_1, \epsilon_2)$$

$$x_3 = f_\theta(\epsilon_1, \epsilon_2, \epsilon_3)$$

9.4 Generative Adversarial Networks (GAN)

There are two parts in this model: a generative model G that captures the data distribution, and a discriminative model D that estimates the probability that a sample came from the training data rather than G.We train the two models simultaneously for estimating generative models. Intuitively, we can think of the generative model as analogous to a team of counterfeiters, trying to produce fake currency and use it without detection, while the discriminative model is analogous to the police, trying to detect the counterfeit currency. Competition in this game drives both teams to improve their methods until the counterfeits are indistiguishable from the genuine articles.

9.4.1 Adversarial Nets

To learn the generator's distribution p_g over data x, we define a prior on input noise variables $p_z(z)$, then represent a mapping to data space as $G(z; _g)$, where G is a differentiable function represented by a multilayer perceptron with parameters $_g$. We also define a second multilayer perceptron $D(x; _d)$ that outputs a single scalar. $D(x)$ represents the probability that x came from the data rather than pg. We train D to maximize the probability of assigning the correct label to both training examples and samples from G. We simultaneously train G to minimize $\log(1-D(G(z)))$. So the loss function is:

$\min_G \max_D V(D, G) = E_{x\sim p_{data}(x)}[logD(x)] + E_{z\sim p_z(z)}[\log(1 - D(G(z)))]$

9.4.2 Mathematical Analysis

For G fixed, the optimal discriminator D is

$D^*_G(x) = \frac{p_{data}(x)}{p_{data}(x)+p_g(x)}$

Note that the training objective for D can be interpreted as maximizing the log-likelihood for estimating the conditional probability $P(Y = y|x)$, where Y indicates whether x comes from p_{data} (with $y = 1$) or from p_g (with $y = 0$).

9.4.3 Objective at Optimal Discriminator

From the optimal discriminator in section 9.4.2 and objective function in section 9.4.1, we can write the objective at Optimal discriminator as follows:

$$L(M,D) = E_{x\sim p_{data}(x)}[logD(x)] + E_{z\sim p_z(z)}[\log(1-D(G(z)))]$$

Substituting optimal discriminator we get,

$$L(M) = E_{x\sim p_{data}(x)}[log\frac{p_{data}(x)}{p_{data}(x)+p_g(x)}] + E_{z\sim p_z(z)}[\log(1-\frac{p_{data}(x)}{p_{data}(x)+p_g(x)})]$$

$$L(M) = E_{x\sim p_{data}(x)}[log\frac{p_{data}(x)}{p_{data}(x)+p_g(x)}] + E_{z\sim p_z(z)}[log\frac{p_g(x)}{p_{data}(x)+p_g(x)}]$$

$$L(M) = E_{x\sim p_{data}(x)}[log\frac{\frac{p_{data}(x)}{2}}{\frac{p_{data}(x)}{2}+\frac{p_g(x)}{2}}] + E_{z\sim p_z(z)}[log\frac{\frac{p_g(x)}{2}}{\frac{p_{data}(x)}{2}+\frac{p_g(x)}{2}}]$$

$$L(M) = -log(4) + E_{x\sim p_{data}(x)}[log\frac{p_{data}(x)}{\frac{p_{data}(x)}{2}+\frac{p_g(x)}{2}}] + E_{z\sim p_z(z)}[log\frac{p_g(x)}{\frac{p_{data}(x)}{2}+\frac{p_g(x)}{2}}]$$

$$L(M) = -log(4) + KL(p_{data}(x)||\frac{p_{data}(x)}{2}+\frac{p_g(x)}{2}) + KL(p_g(x)||\frac{p_{data}(x)}{2}+\frac{p_g(x)}{2})$$

$$L(M) = -log(4) + 2JSD(p_g(x)||p_{data}(x))$$

where JSD is Jenson Shannon Divergence. Since the Jenson Shannon Divergence will have minimum value 0 only when $p_g(x) = p_{data}(x)$, the global minimum value of the objective will be $-log(4)$ and it will be attained only when $p_g(x) = p_{data}(x)$. Thus we can conclude that generative model must replicate the data generating process for the optimal results.

10 4/5/2022 Causal Inference

The goal of this lecture is to understand how interventions are different from modelling. Putting this in the context of healthcare, we might want to see the effect of Medication A vs Medication B on preventing a heart attack. In this case, giving a medication is an 'intervention', and causal inference seeks to estimate the effect of this intervention. Here are some other examples of questions that we might be interested in answering:

- Which statin to give to a patient with hyperlipidemia?
- Which medication to give to a depressed person?
- Which patients to give hospice care to?

One solution might be to find important features when we pose this as a regression problem, but without developing a mathematical framework of cause and effect it is difficult to formulate a solution.

10.1 Potential Outcomes

We use $Y_{n,i}$ to define the outcome of treatment n on subject i. As an example, $Y_{0,i}$ might be the PHQ-9 score after 2 months after starting medication for depression, and $Y_{1,i}$ might be the PHQ-9 score after 2 months after starting cognitive behavioral therapy for depression.

10.2 Fundamental Problem of Causal Inference

Once we have these potential outcomes, we are interested in aswering questions like "Is treatment A better than treatment B?" ($Y_{0,i} > Y_{1,i}$) or "What's the difference between A and B?" ($Y_{0,i} - Y_{1,i}$). One solution to answer these questions might be to observe these outcomes ($Y_{0,i}, Y_{1,i}$), identify interesting features (x_i), build a model for $Y_{0,i}, Y_{1,i}$ and x_i, and finally make predictions. But this approach is limited by the Fundamental Problem of Causal Inference [Holland, 1986]. We cannot observe the effect of both treatments on the same person. We can, however, observe the population level effects.

10.3 Population Effects

We define the Average Treatment Effect (ATE) as the expected value of the difference in potential outcomes. The Individualised Treatment Effect (also called Conditional Average Treatment Effect) is the ATE conditioned on covariates x_i. It is easy to see that ATE is equivalent to ITE marginalized over covariates. The central idea is to use different individuals to compute population level estimates.

$$\begin{aligned}
\text{ATE} &= \mathbb{E}[y_{1,i}] - \mathbb{E}[y_{0,i}] \\
\text{ITE}(x_i) &= \mathbb{E}[y_{1,i}|x_i] - \mathbb{E}[y_{0,i}|x_i] \\
\text{ATE} &= \mathbb{E}_{p(x_i)}[\text{ITE}(x_i)]
\end{aligned}$$

10.4 Terminology

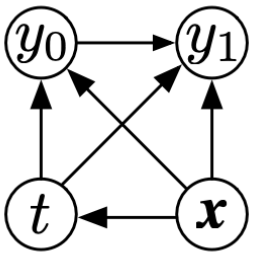
Let t refer to the treatment of an individual, where $t = 0$ refers to receiving no treatment (control) and $t = 1$ refers to receiving a particular treatment. We also use $y_{1-t,i}$ to define the counterfactual, which basically refers to the unobserved potential outcome. So if $t = 0$, the counterfactual will be $y_{0,i}$, and if $t = 1$ the counterfactual will be $y_{1,i}$.

10.5 Randomization

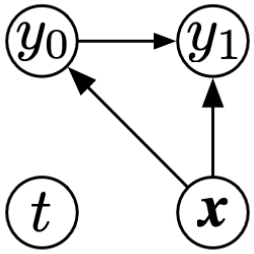
Based on this terminology, one might compute the ATE by computing the expectations over the observed outcomes. However, here's the problem in this approach:

$$\mathbb{E}[y_{1,i}|t = 1] - \mathbb{E}[y_{1,i}|t = 1] \neq \mathbb{E}[y_{1,i}] - \mathbb{E}[y_{1,i}]$$

We only get to observe $y_{1,i}$ for the part of the population which has been exposed to the treatment $t = 1$. In other words, the people that get treated might just be sicker, older, or different from the



(a) The assignment of t is dependent upon x_i



(b) The assignment of t is random, independent of x_i

Figure 19: The effect of randomization on the computational graph of y

control population. This can be simply because the people who receive treatments are the ones who really need them. The ones who don't need treatment don't receive any.

To get away with this problem, we can just assign treatments randomly, which breaks dependence. Figure 19 graphically represents this idea. By introducing randomness, we can find the ATE by computing the expectations over the observed outcomes. Similarly, we can also calculate the ITE.

$$\mathbb{E}[y_{1,i}|t=1]-\mathbb{E}[y_{1,i}|t=1]=\mathbb{E}[y_{1,i}]-\mathbb{E}[y_{1,i}]$$

This, however, poses another problem. it is not always possible to conduct a randomized trial because often times they are hard, expensive or unethical to run.

10.6 Observational Causal Inference

We move on to estimating the same goal of observing the difference in the average outcomes i.e $\mathbb{E}[y_1]-\mathbb{E}[y_0]$. However, now we can no longer make the assumption that the outcomes are independent of the treatment t.

$$y_1, y_0 \not\perp\!\!\!\perp t$$

In general, the outcomes are not independent of the treatment obtained. For example, a treatment decision based on how sick a person is with the output being whether they live or not, averaging over the degree of sickness would show a dependence between the treatment and result (whether you live). Estimating the causal effect is not possible without some independence assumption.
An assumption we can make is that the outcomes y_1, y_0 are conditionally independent of the treatment t given a set of covariates $\mathbf{x}$. This is called "Strong Ignorability".

10.7 Strong Ignorability

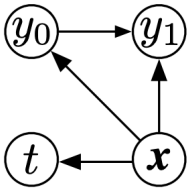


Figure 20: Graph representing conditional independence of outcomes from treatment given covariate

We can now simply condition on $\mathbf{x}$ first:

$$\mathbb{E}[y_1] - \mathbb{E}[y_0] = \mathbb{E}_x[\mathbb{E}[y_1|x]] - \mathbb{E}_x[\mathbb{E}[y_0|x]]$$

Now that $\mathbf{x}$ is observed, t, y_i are independent and we can add it to the output:

$$= \mathbb{E}_x[\mathbb{E}[y_1|x, t=1]] - \mathbb{E}_x[\mathbb{E}[y_0|x, t=0]]$$

Both the expectations are values that we can now compute:

- $\mathbb{E}[y_1|x, t=1]$ is the average treated outcome on treated
- $\mathbb{E}[y_1|x, t=0]$ is the average untreated outcome on untreated

that is assuming a discrete set of covariates, say age, we can average the output over the target treated/untreated set.

$$y_1, y_0 \perp\!\!\!\perp t|x$$

Intuitively, strong ignorability can be a case of randomization albeit conditional randomization, since a fixed value of $\mathbf{x}$ in the computation graph of Fig. 20, then t becomes disconnected and hence independent from y_0, y_1. In contrast to the randomization graph, where t is decided by independent coin flips, there is now a bias for the coin associated with certain covariates, say bias towards heads is more with increasing age, and hence is a case of conditional randomization.

10.7.1 Problems

- We cannot check Strong Ignorabilty: that is the condition that y_1, y_0 are independent of t given x. This is because y_1 and y_0 for a given covariate x is never observed together, it is either y_1 or y_0, hence the marginals of the outcomes cannot be fully computed.
- When support of the covariates x given treatment $t=0$ and $t=1$ are disjoint, we cannot say whether the actual covariate matters or the treatment.

  $$Supp(p(x|t=1)) \cap Supp(p(x|t=0) = \emptyset$$

  For example, if treatment if only given to people whose age > 50, then the conditional of $y|x, t$ is confusing since the model would not know whether it is the age that matters or the treatment. Considering the **ATE**, we cannot distinguish between x and t since t determines part of x.
  Hence, full support is needed for strong ignorability.

10.8 The Functional View

In a broad sense, the world can be represented in the form of functions such as

$$\mathbf{x} = f(\epsilon_x)$$
$$t = g(\mathbf{x}, \epsilon_t)$$
$$y = h(t, \mathbf{x}, \epsilon_y)$$

where $\epsilon_x, \epsilon_t, \epsilon_y$ are jointly independent.

In laymen terms, it means that we can treat the observations around us $\mathbf{x}$ as a function over some noise. We can then derive the values for t which can be thought of as $p(t|\mathbf{x})$. Based on our observations of $\mathbf{x}$ and t, we can obtain the outcomes for y or $p(y|t, \mathbf{x})$.

From this distribution of the world, we can write the conditional expectation as,

$$\mathbb{E}[y|t=5] = \mathbb{E}_{p(\epsilon_y, \mathbf{x}|t=5)}[h(5, \mathbf{x}, \epsilon_y)]$$

Since we know that ϵ_y is sampled independently of t,

$$\mathbb{E}[y|t=5] = \mathbb{E}_{p(\mathbf{x}|t=5)p(\epsilon_y)}[h(5, \mathbf{x}, \epsilon_y)]$$

We can get rid of the dependency of t over $\mathbf{x}$ as shown in Figure 20 by intervening on t. We can write the intervened distribution of the world as,

$$\mathbf{x} = f(\epsilon_x)$$

$$t = 5$$
$$y = h(t, \mathbf{x}, \epsilon_y)$$

The conditional expectation now can be written as,

$$\mathbb{E}[y|t=5] = \mathbb{E}_{p(\mathbf{x})p(\epsilon_y)}[h(5, \mathbf{x}, \epsilon_y)]$$

We can make a slightly more realistic distribution by using a randomized distribution rather than an intervened distribution, which can be represented as,

$$\mathbf{x} = f(\epsilon_x)$$
$$t = g(\epsilon_t)$$
$$y = h(t, \mathbf{x}, \epsilon_y)$$

The conditional expectation of this distribution can also be written as

$$\mathbb{E}[y|t=5] = \mathbb{E}_{p(\mathbf{x})p(\epsilon_y)}[h(5, \mathbf{x}, \epsilon_y)]$$

because t is independent of $\mathbf{x}$

10.9 Propensity scores

From strong ignorability,

$$y_1, y_0 \perp\!\!\!\perp t|x$$

Hence, the chance that you are treated depends on x Estimate of $p(t=1|x)$ is called propensity score. We can compute this by using may models like nn and logistic regression etc. After we can find this, we can use importance sampling to estimate the expectations of the outcome of the various groups.

$$ATE = \mathbb{E}_{p(x)}[\mathbb{E}[y_1|x]] - \mathbb{E}_{p(x)}[\mathbb{E}[y_0|x]]$$
$$= \int p(x)\frac{p(t=1|x)}{p(t=1|x)}\mathbb{E}[y_1|x]dx - \int p(x)\frac{1-p(t=1|x)}{1-p(t=1|x)}\mathbb{E}[y_0|x]dx$$
$$= \int p(x)\frac{\mathbb{E}[t|x]}{p(t=1|x)}\mathbb{E}[y_1|x]dx - \int p(x)\frac{\mathbb{E}[1-t|x]}{1-p(t=1|x)}\mathbb{E}[y_0|x]dx$$

Here t and y_0, y_1 given x are independent so we can multiply expectations

$$= \int p(x)\frac{\mathbb{E}[ty_1|x]}{p(t=1|x)}dx - \int p(x)\frac{\mathbb{E}[y_0(1-t)|x]}{1-p(t=1|x)}dx$$
$$= \int p(x)\frac{\mathbb{E}[ty|x]}{p(t=1|x)}dx - \int p(x)\frac{\mathbb{E}[y(1-t)|x]}{1-p(t=1|x)}dx$$
$$= \int p(x)\frac{\int typ(t,y|x)dtdy}{p(t=1|x)}dx - \int p(x)\frac{\int(1-t)yp(t,y|x)dtdy}{1-p(t=1|x)}dx$$
$$= \int \frac{typ(x)p(t,y|x)}{p(t=1|x)}dtdxdy - \int \frac{(1-t)yp(x)p(t,y|x)}{1-p(t=1|x)}dtdxdy$$
$$= \int \frac{typ(x,t,y)}{p(t=1|x)}dtdxdy - \int \frac{(1-t)yp(x,t,y)}{1-p(t=1|x)}dtdxdy$$

We can see that, if the chance of treatment is really small for fixed value of x then the variance is high. We can also combine propensity scores and regression to improve the predictability and in turn it decreases bias and variance

Evaluation of the causal inference treatment effects is often affected by the unobserved confounding variables. Hence, we use the sensitivity analysis to evaluate the causal inference drawn from the treatment.

11 4/12/2022 Reinforcement Learning

The goal of this lecture is to take actions to reach a goal, with actions depend on the state.

11.1 Markov Decision Processes

A Markov Decision Process is an extension of the standard (unhidden) Markov model. Each state has a collection of actions that can be performed in that particular state. These actions serve to move the system into a new state. More formally, the MDP's state transitions can be described by the transition function $T(s, a, s')$, where a is an action moving performable during the current state s, and s' is some new state.

As the name implies, all MDPs obey the *Markov property*, which holds that the probability of finding the system in a given state is dependent only on the previous state. Thus, the system's state at any given time is determined solely by the transition function and the action taken during the previous timestep:

$$P(S_t = s' | S_{t-1} = s,\ a_t = a) = T(s, a, s')$$

Each MDP also has a reward function $R : S \mapsto \mathbb{R}$. This reward function assigns some value $R(s)$ to being in the state $s \in S$. The goal of a Markov Decision Process is to move from the current state s to some final state in a way that a) maximizes $R(s)$ and b) maximizes R's potential value in the future.

Given a Markov Decision Process we wish to find a *policy* – a mapping from states to actions. The policy function $\Pi : S \mapsto A$ selects the appropriate action $a \in A$ given the current state $s \in S$.

11.2 Value Iteration

The consequences of actions (i.e., rewards) and the effects of policies are not always known immediately. As such, we need some mechanisms to control and adjust policy when the rewards of the current state space are uncertain. These mechanisms are collectively referred to as *reinforcement learning*.

One common way of trading off present reward against future reward is by introducing a *discount rate* γ. The discount rate is between 0 and 1, and we can use it to construct a weighted estimate of future rewards:

$$\sum_{t=0}^{\infty} \gamma^t r_t$$

Here, we assume that $t = 0$ is the current time. Since $0 < \gamma < 1$, greater values of t (indicating rewards farther in the future) are given smaller weight than rewards in the nearer future.

Let $V^{\Pi}(s)$ be the value function for the policy Π. This function $V^{\Pi} : S \mapsto \mathbb{R}$ maps the application of Π to some state $s \in S$ to some reward value. Assuming the system starts in state s_0, we would expect the system to have the value

$$V^{\Pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t) \mid s_0 = s, \Pi\right]$$

Since the probability of the system being in a given state $s' \in S$ is determined by the transition function $T(s, a, s')$, we can rewrite the formula above for some arbitrary state $s \in S$ as

$$V^{\Pi}(s) = R(s) + \sum_{s'} T(s, a, s') \gamma V^{\Pi}(s')$$

where $a = \Pi(s)$ is the action selected by the policy for the given state s.

Our goal here is to determine the optimal policy Π^*(s). Examining the formula above, we see that $R(s)$ is unaffected by choice of policy. This makes sense because at any given state s, local reward

term $R(s)$ is determined simply by virtue of the fact that the system is in state s. Thus, if we wish to find the maximum policy value function (and therefore find the optimum policy) we must find the action a that maximizes the summation term above:

$$V^{\Pi^*}(s) = R(s) + \max_a \sum_{s'} T(s,a,s')\gamma V^{\Pi^*}(s')$$

Note that this formulation assumes that the number of states is finite.

The formula above forms the basis of the *value iteration* algorithm. This operation starts with some initial policy value function guess and iteratively refines $V(s)$ until some acceptable convergence is reached:

Algorithm 1 Value Iteration

1. Initialize $V(s)$.
2. Repeat until converged:
   (a) for all $s \in S$:
      i. $R(s) = R(s) + \max_a \sum_{s'} T(s,a,s')\gamma V(s')$
      ii. $\Pi(s) = \arg\max_a Q(s,a)$

Each pass of the value iteration maximizes $V(s)$ and assigns to $\Pi^*(s)$ the action a that maximizes $V(s)$. The function $Q(s,a)$ represents the potential value for $V(s)$ produced by the action $a \in A$.

11.3 Q-Learning

The example of value iteration above presumes that the transition function $T(s,a,s')$ is known. If the transition function is not known, then the function $Q(s,a)$ can be obtained through a similar process of iterative learning, the aptly-named *Q-learning*.

The naive guess for a Q-learning formula would be one that closely resembles the policy value function, such as

$$Q(s,a) = R(s) + \gamma\left[\max_{a'} Q_{old}(s,a')\right]$$

This formula aggressively replaces old values of Q, though, which is not always desirable. For better results [1], use a weighted average learning function:

$$Q(s,a) = \eta Q_{old}(s) + (1-\eta)\gamma\left[\max_{a'} Q_{old}(s,a')\right]$$

where η is a user-selected learning parameter.

This formula can be turned into an algorithm much like the value iteration algorithm above. The only difference is that the action is selected by a user-defined function $f(s)$, which returns the appropriate policy action $\Pi(s)$ most of the time, but occasionally selects a random action to blunt the effects of sampling bias.

Algorithm 2 One-Step Q-Learning

1. Initialize $\Pi(s)$ to $\arg\max_a Q(s,a)$.
2. Repeat until Π converges:
   (a) For each $s \in S$:
      i. Select an action $a = f(s)$.
      ii. $Q(s,a) = \eta Q_{old}(s) + (1-\eta)\gamma\left[\max_{a'} Q_{old}(s,a')\right]$
      iii. $\Pi(s) = \arg\max_a Q(s,a)$.

11.4 Policy Gradients

The goal is to maximize the expected reward of a trajectory

$$\mathscr{L}(\theta) = \mathop{\mathbb{E}}_{\tau \sim \pi_\theta}[r_\tau]$$

where r_τ is reward of the trajectory,

$$r_\tau = \sum_{t=1}^{T} \gamma_t r_t$$

Using tools from variational inference, optimize this objective by using *score function gradients*.

$$\begin{aligned}\nabla_\theta \mathscr{L}(\theta) &= \nabla_\theta \mathop{\mathbb{E}}_{\tau \sim \pi_\theta}[r_\tau] \\ &= \mathop{\mathbb{E}}_{\tau \sim \pi_\theta}[\nabla_\theta \log \pi_\theta(\tau) r_\tau]\end{aligned}$$

Get a noisy unbiased estimate of the gradient of expectation using Monte Carlo methods.

$$\hat{\nabla}_\theta \mathscr{L}(\theta) = \frac{1}{S} \sum_{s=1}^{S} \nabla_\theta \log \pi_\theta(\tau) r_\tau$$

Maximize using stochastic optimization.

Problem: High variance (which is bad for stochastic optimization)

To control the variance, you can Rao Blackwellize, or introduce a baseline.

12 4/19/2022 Cautionary Tales in Machine Learning

The goal of this lecture is to provide a brief overview of the topics and ideas covered in the course and to address common problems with applying machine learning models to real-world applications.

12.1 Topics Covered in the Course

12.1.1 Supervised Learning

12.1.2 Bayesian Methods

12.1.3 Graphical/Latent Variable Models

12.1.4 Causal Inference

12.1.5 Reinforcement Learning

12.2 How Does it Work?

A trained machine learning model is dependent on both the training data and the structure of the model used. To make any inference on an input that isn't represented in the training data fundamentally requires assumptions, even if the unseen input is very similar to a input present in the training data.

12.3 Adversarial Examples

Since machine learning model outputs are highly dependent on the input and loss functions are generally highly irregular, adjusting an image's pixels can produce a completely different classification. Producing the noise using the gradient of the loss with respect to the input can find the locally optimal way to shift the pixel values of the input image to increase the loss, requiring a very small shift to produce an incorrect output. As we see in Figure 21, the shift is too small for the human eye to perceive. This is especially effective on high-dimensional data like images, since the gradient can be divided among the many dimensions.

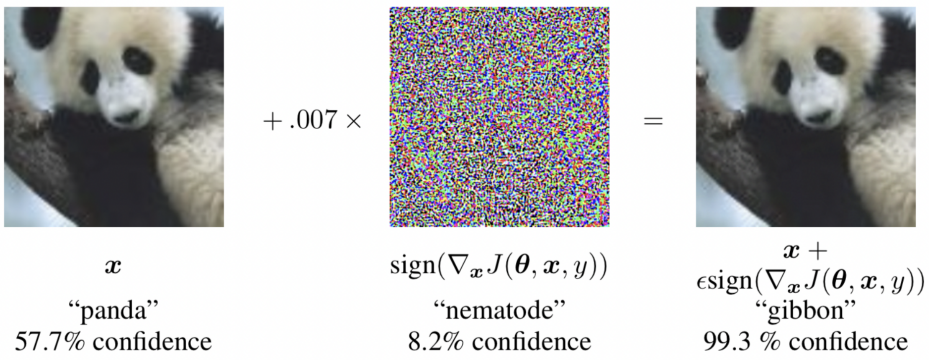


Figure 21: Adjusting the input image by a small amount of adversarial noise can drastically change its classification

Method to create an adversarial example:

1. Start with a trained model
2. Compute the gradient of the loss function with respect to the input
3. Follow the gradient in the direction of increasing loss
4. Limit the movement so the new input is still visually similar

In some applications like electrocardiograms (ECG) that track heart rate, adversarial examples can have sharp perturbations evident of an altered ECG. However, this can be fixed by smoothing the adversarial noise to be indistinguishable from human-produced ECGs, even to expert doctors.

12.4 Model Not Good Enough

CLIP is a language/image model released by Open AI. It was trained on over 400 million images to predict captions associated with an image. It typically performs very well on most images, but in Figure 22, putting a slip of paper saying "iPod" on a granny smith apple is enough to convince the model that the image is of an iPod. This is a mistake that no human would make, but the model is very certain of its classification.

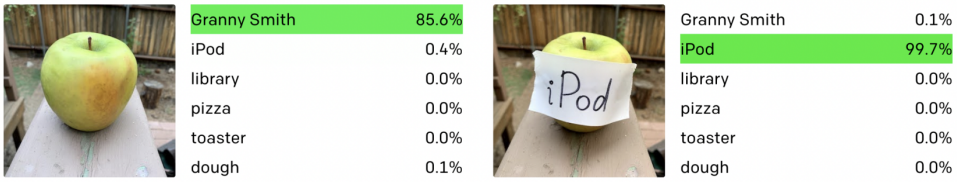


Figure 22: You can convince the model to classify an apple as an iPod by writing "iPod" on a slip of paper

To train a classifier $p(y|\mathbf{x})$, it only exists when $p(\mathbf{x}) > 0$ since $p(y|\mathbf{x}) = \frac{p(y,\mathbf{x})}{p(\mathbf{x})}$. Although it's possible that the image of the apple with "iPod" written on it exists in the training data, it's extremely unlikely. Therefore the model has to extrapolate, making assumptions about the unseen image. In this case, the model is probably assuming that any text present in an image is descriptive of the object(s) in the image. The problem is this isn't necessarily a bad assumption since it's true for the majority of images in the training data. How do we train models to perform well on unseen data when correct assumptions on the training data don't hold in general?

12.5 Importance of label selection

Insurance asks to build models to reduce the costs. The model takes in data and predicts the costs.The idea is to give early attention to the people with predicted high costs than with lower predicted costs. But this algorithm shows a significant racial bias as in 23 and 24. It shows lower costs prediction for black people at the same chronic illness level as white people.

The algorithm gives the same risk score for black people at a higher sickness level than white people as shown in 24.This significantly affects the help received by black people.

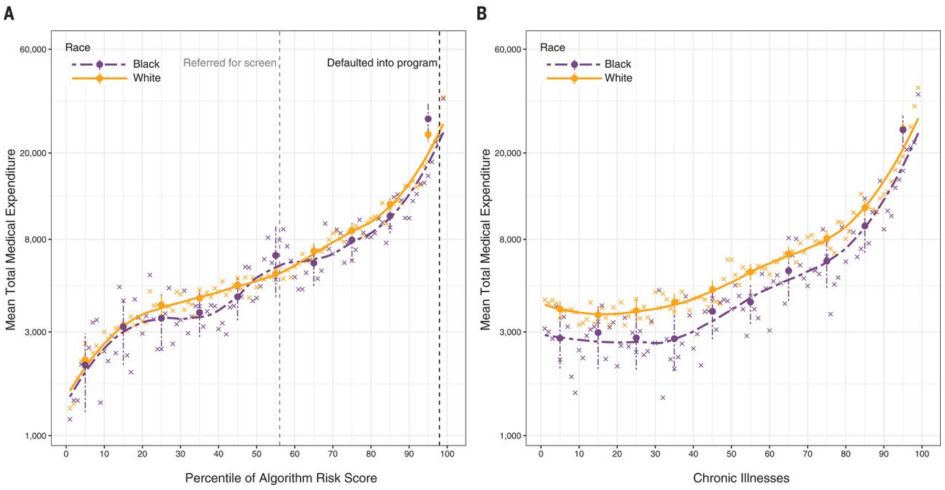


Figure 23

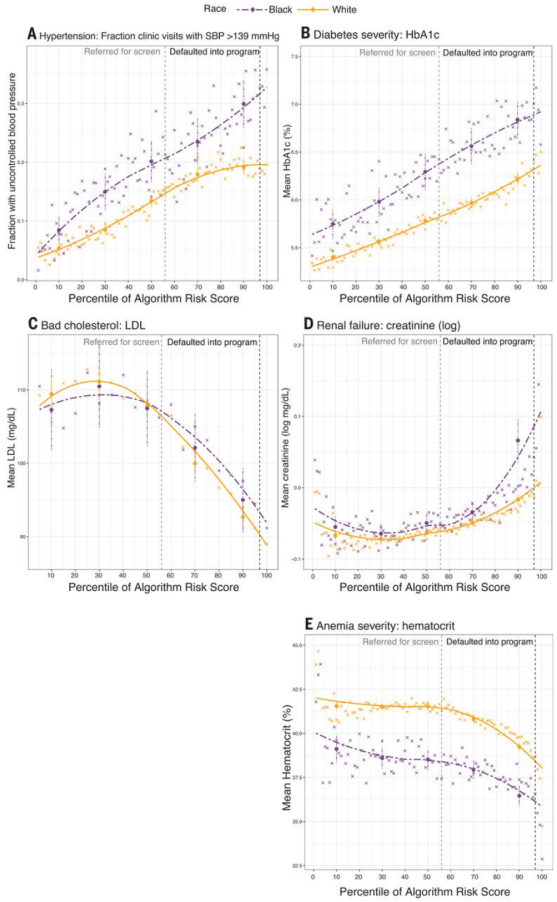


Figure 24

Although, the sickness level is same, the cost utilization may differ across groups. Obermeyer et. al (2019) say that black people might have lesser visits to the emergency room or to the doctor's office compared to white people due to several reasons - direct discrimination, doctor-patient relationship, less trust of black people in healthcare or doctors according to a survey, etc. As the objective is to reduce costs, the algorithm takes the cost utilization in the previous years and predicts the future costs. This creates the bias. This can be corrected by predicting health level - like chronic illnesses instead of predicting costs. Thus, label selection should be done carefully as it can introduce bias into a model.

12.6 The methodology

Growth of published reinforcement learning papers.Shown are the number of RL-related publications (y-axis) per year (x-axis) scraped from Google Scholar searches.

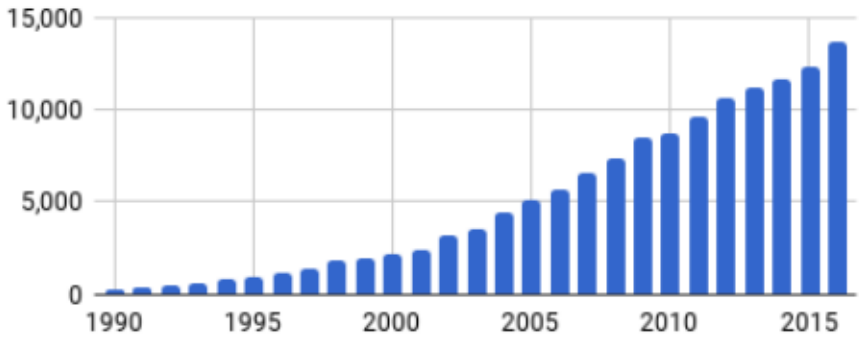


Figure 25

Performance of several policy gradient algorithms across benchmark MuJoCo environment suites

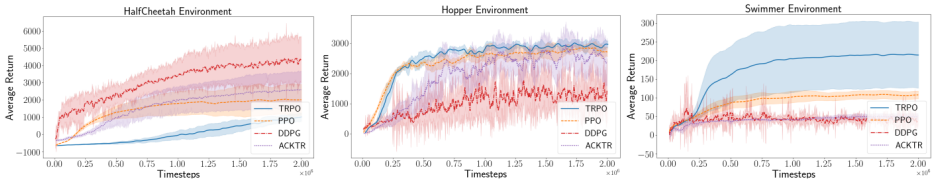


Figure 26

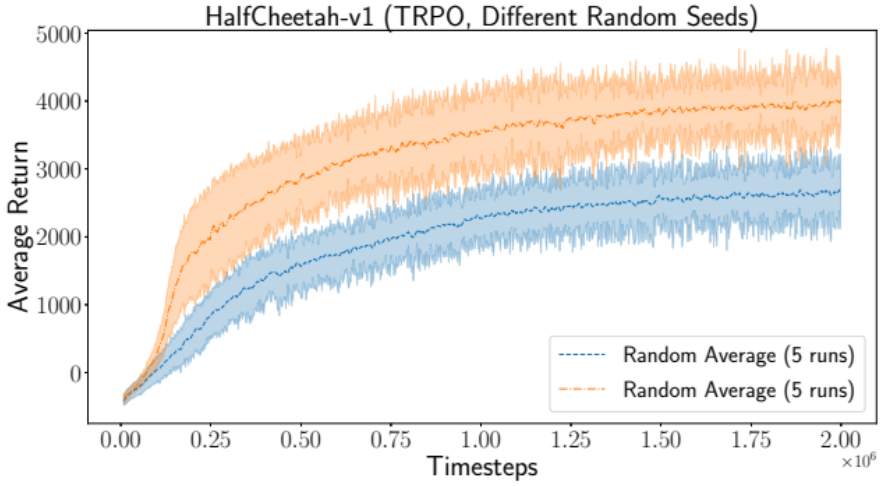


Figure 27

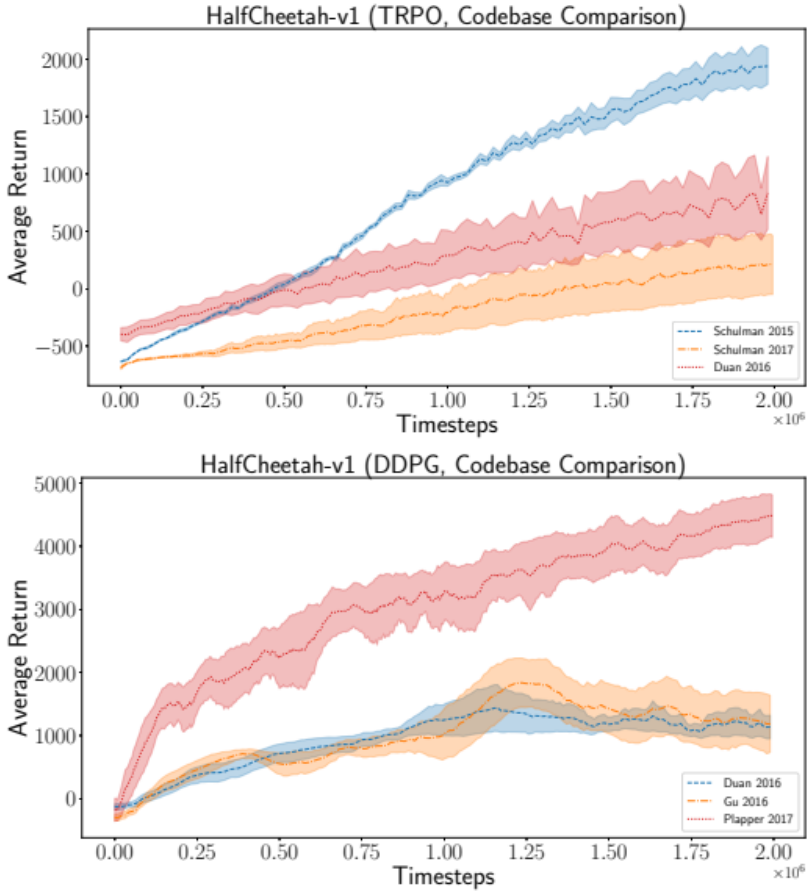


Figure 28

TRPO codebase comparison using our default set of hyperparameters (as used in other experiments)

13 4/26/2022 Concept Reviews (Victory Lap)

Today's lecture is an overview of the material covered throughout the course.

13.1 What we've learned

- Supervised Learning
- Latent Variable Models/Deep Generative Models
- Causal Inference
- Reinforcement Learning

13.2 Missing Data

- Normally we won't have all the data in a class
  - just train models directly?
  - fill in values with zeros?
  - fill in values with mean?
- Missingness may come from a data generating process with x as features, y as a label, and m as a missingness indicator with dimension $\dim(x)+1$.
- Even with infinite data, we need more assumptions to estimate $p(y|x)$ since we can never estimate $p(m_y = 0|x)$. Therefore, we make more assumptions:
- – Assume m only depends on features, which can be expressed as $P(m,x,y) = P(x)P(y|x)P(m|x)$, then $P\left(y|x, \left(\left[\sum_{j=1}^{dim(x)+1} m_j\right] = 0\right)\right) = P(y|x)$.
  Note: m is a vector, m's sum is zero when nothing is missing.
  - Assume m only depends on non-missing data
  - Assume m is random and model the randomness
- Can filling in data improve predictions? Consider the data processing chain,

$$y \rightarrow (x_{observed}, m) \rightarrow (x_{observed}, m, x_{imputed}) \rightarrow (x_{observed}, x_{imputed})$$

by data processing inequality:

$$I[y; x_{observed}, m] \geq I[y; x_{observed}, x_{imputed}]$$

No, as a purely logical consequence, via the data processing inequality.

13.3 Embedding

People try to represent anything discrete with vectors.

Vectors for an object should be

- close in inner product to related objects
- far in inner product to unrelated objects

An example objective:

$$\mathscr{L}_i = \sum_{j \in RelatedObjects} \log \sigma(x_i^T x_j) + \sum_{j \in UnrelatedObjects} \log \sigma(-x_i^T x_k) \tag{82}$$

Totaling

$$max \mathscr{L} = \sum_i \mathscr{L}_i \tag{83}$$

Can use separate embeddings for an object in $\mathscr{L}_i$.

Where do we get vectors? Here is an example:

- Assign all songs as a vector
- Related songs are songs that appear in a playlist together
- Unrelated songs are random songs
- Users are the average of the songs they listen to

This example is close to some basic internals at companies and we can see that we have a lot of design choices based on our data and applications (e.g. objects have a graph like in Facebook).

13.4 Time Series

A lot of real world info come in sequences over time, e.g. financial sequences, user data, images on Instagram. Fundamentally, time is always progressing, and more data comes in with time. There are various ML models available to model such time-dependent data (time series).

Classes of Time Series Models:

- *General Autoregressive Models*: Predicts future behavior based on past behavior
  - ARIMA models: Autoregressive Integrated Moving Average Model
  - Markov chains, Linear Dynamic Systems
- *Gaussian Process*: An infinite dimensional generalization of a Gaussian distribution
- *Neural Networks*: Composed by many neurons which are connected together in complex interconnections to solve linear or non-linear problems
  - Recurrent Neural Networks (RNNs)
  - Dilated Convolutional Neural Networks
  - Attention

13.5 Real World Example

Scale from perception to causation increases noise/assumptions as you go up the scale. The world is changing, we need to check the data as time goes on. An example of a way to do this would be:

- Conditional Randomization Testing which creates fake data and then compares it with real data

13.6 How to Deploy/Convey Model

- Defining the problem
- Collecting the data
- Checking/Processing the data
- Building models
- Communicating results
- Monitoring after deployment

References

P. W. Holland. Statistics and causal inference. *Journal of the American statistical Association*, 81 (396):945–960, 1986.

Y. LeCun, C. Cortes, and C. Burges. Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010.

A an appendix section

...