CS3244 Cheatsheet

Evaluation metrics – Accuracy, AUC-ROC curve (tells how well the classifier is able to separate positive from negative classes), Log-loss

Intro to Machine Learning

The ML Pipeline

Data -> Modeling -> Insights/Inferences

Model – A mathematical representation of a behaviour
(Via linear, non linear eqns, trees …)

Iterative process of:
1. Data Collection
2. Data Extraction (Feature engrng)
3. Data understanding (w visualstn)
4. Data pre-processing
5. Model choice/design
6. Model training
7. Model validation (evaluation)
8. Model understanding (visualization / explainability)
9. Model deployment

Feature Engrng – Process of using domain knowledge to extract features from raw data

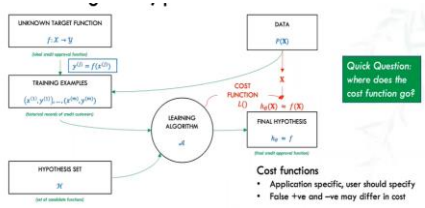
Paradigms of ML

3 generic variations:
1. Supervised learning
- Each input comes with corresponding label
2. Unsupervised learning
- Identify patterns in datasets that are neither classified nor labelled
3. Reinforcement learning
- Reward desired behaviours and/or punishing undesired ones

Components of Learning

- Input x → output y
- target fn: f: x → y
- hypothesis: h: x → y

- Target function refers to what is fixed but unknown (i.e. the true function), h is what we hypothesise based on what we learnt from D, the dataset.



2 components of a learning algo:

1. The *Hypothesis set* $\mathcal{H} = \{h_1, h_2, \dots, h_{|\mathcal{H}|}\}$
2. The *Learning algorithm* $\mathcal{A}$ selects $h \in \mathcal{H}$

Together referred to as learning model. We choose H and learning algo A chooses h.

Data matrix – illustrated as a matrix of m rows and n columns, where each column is a feature

Parameters – To pick the best h, our parameters are tuned. Parameters for a linear model are like the gradient and intercept.

Unsupervised learning

- Data is not labelled
→ Clustering, grouping, density estimation, outlier detection, noise filtering

Reinforcement Learning

- Guidance on our behaviours instead of (input, correct output)
→ No supervisor but a reward signal
→Delayed feedback
→ Agents actions affect the subsequent data received
e.g. playing moba, doing stunts on helicopter, alphago

Inductive Bias

→ Set of assumptions a learner uses to predict outputs of given inputs it has not encountered
- A learner must have inductive bias to make learning feasible, else it has no rational basis for classifying data it hasn't seen before

Five ML Tribes

→ Symbolists, Analogizers, Bayesians, Connectionists, Evolutionaries
Visualised by 3 parameters:
- Representation: How algo associated with each tribe can be represented
- Evaluation: How algorithm is evaluated depending on tribe it belongs to
- Optimisation/Solve: How an algo can be optimised to achieve optimum result

Symbolists

Solve: Inverse Deduction
Evaluate: Accuracy
Representation: Logic

- Focus on symbol manipulation where questions can be presented as eqns
- Inverse deduction – Starting with certain knowledge and exponentially getting closer to label by answering questions.
→ Decision Tree
→Optimisation using inverse deduction: pruning

Connectionists

Solve: Gradient Descent
Evaluate: Loss Function
Representation: Neural Network

- Follows concepts of how human brain functions, build perceptrons
→ Convolutional Neural Networks (CNN), linear/logistic regression
→ Loss Function/squared error evaluation

Evolutionaries

Solve: Genetic Search
Evaluate: Fitness / Reward
Representation: Genetic Programs

→ Follow enhancement by natural selection, known as coined crossover and mutation.
→ Fitness – Choose best solution from pool of solutions

Bayesian

Solve: Probabilistic Inference
Evaluate: Posterior Probability
Representation: Graphical Models

- Consider an event occurrence and predict that one of the known causes was the contributing factor.
- graphs as representation
- probabilistic inference – Task of deriving probability of one or more random variables taking a specific value or set of values

Analogizers

Solve: Constrained Optimization
Evaluate: Margin
Representation: Support Vectors

- Identify similarities between situations and things, using metrics such as Manhattan/Euclidean distance
→ KNN, SVM

5 tribes → different set of assumptions about data, and a tribe's representation may be more suited to a problem's setting

K-NN

Compare something with something else to assist understanding
Train: Memorize data – O(1) time
Test: Compute which training instances are closest for m features – O(m * n)

Distance Metric

Different ways this can be done:
- L1 distance (Manhattan)
- L2 distance (Euclidean)

L1 distance metric: $d(x^{(*)}, x^{(1)}) = \sum_p \left| x_p^{(*)} - x_p^{(1)} \right|$

Take the difference between each feature pairwise of the test data point and training datapoint, sum them up

KNN Regression

- Test instance predict mean of its K-NN

→ We often use L2 (Euclidean) on our first try

$$L2(x^{(*)}, x^{(i)}) = \sqrt{\left(x_1^{(*)} - x_1^{(i)}\right)^2 + \left(x_2^{(*)} - x_2^{(i)}\right)^2 + \cdots + \left(x_n^{(*)} - x_n^{(i)}\right)^2}$$

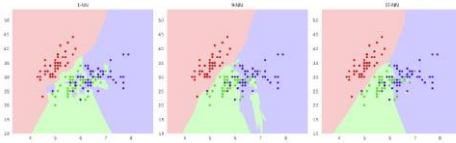
Take the difference between each feature pairwise, square, sum them up then root

→ Any distance metric is possible (non-negativity, symmetric, obeys triangle inequality..) possible, define distance metric to fit your data's semantic meaning!
→ Performs BADLY on images.

These 4 images are very different, but have the same L1 distance! Because it is simply summed up

Features

- Model free algorithm
- Normalization might improve results
- Hyperparameters: Distance metric and k
- Can use some data reserved as validation set
- Very slow to apply, since we need to do knn on every datapoint
- Doesn't work well in high dimensions (curse of high dimensionality)
- Set k as odd number to reduce ties. Ties can still commonly happen when k = multiple of number of classes
- Smaller k – more complex surface
- larger k – smoother surface
- Overfitting occurs less with higher k (small k cannot generalize well)
- Features need to be normalized, else they will be dominated by the feature with big values (e.g. salary dominates age in deciding credit problem)



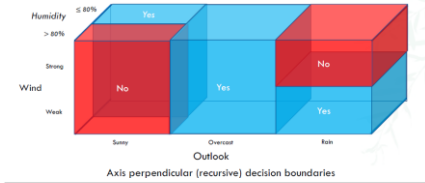
^ 1NN, 9NN, 17NN
- with higher k, anomalous nearest points in training set gets 'overpowered' by other points; its voting power reduces

Maximizing k on **TRAINING** data: 1NN, but if it does not generalize it's a bad h!

Decision Trees

→ Embodies the paradigm of focusing on 1 or 2 key features in making decisions
→ Internal nodes are tests. A node tests exactly 1 component of x (a feature)
→ A branch in the tree corresponds to a test result. Corresponds to an attribute value or a range of attribute values.
→ Each leaf node assigns a class y for classification trees, or real value for regression trees **(trees can do regression!)**
→ **Axis perpendicular (recursive) decision boundaries**: We are looking at a variable (feature), and cutting on an axis. We don't look at a combination of variables, but taking one at a time

Inductive Bias – Decision boundary



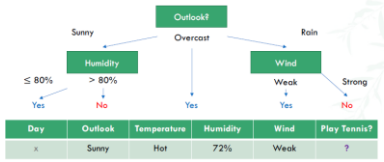
Axis perpendicular (recursive) decision boundaries

```
function DTL(examples, attributes, default) returns a decision tree
  if examples is empty then return default
  else if all examples have the same classification then return the classification
  else if attributes is empty then return MODE(examples)
  else
      best ← CHOOSE-ATTRIBUTE(attributes, examples)
      tree ← a new decision tree with root test best
      for each value v_i of best do
          examples_i ← {elements of examples with best = v_i}
          subtree ← DTL(examples_i, attributes − best, MODE(examples))
          add a branch to tree with label v_i and subtree subtree
      return tree
```

→ DT learner based on examples (training data), attributes, default (inductive bias?).
→ Base cases: No examples? Return default, all examples same classification return classification, no features then return the majority classification of examples (mode).
→ Choose best attribute (based on IG). Create new tree for best attribute, split by its discrete values. (Weather -> rain/sunny/fair), split examples of that best attribute. Recursively call DTL again

DT Considerations

Things to consider:
→ How to pick what to split on(IG/Entropy)
→ How to **discretize** continuous features
→ When do we stop pruning trees (increase generalization!)



Entropy

→ We want our subproblems to be as pure as possible. How to quantify? **Entropy**

$$H(X) = -\sum_{i=0}^{C} P_i \log_2(P_i)$$

Example: for a training set containing p positive examples and n negative examples:

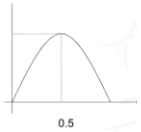
$$H\left(\frac{p}{p+n}, \frac{n}{p+n}\right) = -\frac{p}{p+n}log_2(\frac{p}{p+n}) - \frac{n}{p+n}log_2(\frac{n}{p+n})$$

→ P_i = probability. The lower the probability of P_i → $Log_2(P_i)$ tends towards negative Infinity, multiplied by P_i, we get 0

Entropy curve

For $\frac{p}{p+n}$, the 2-class entropy is:

0 when $\frac{p}{p+n}$ is 0
1 when $\frac{p}{p+n}$ is 0.5
0 when $\frac{p}{p+n}$ is 1



For each produced label in attribute (produced subtree), probability = freq / total;
dataEntropy -= ((probability) * np.log2(probability));
→ Entropy is calculated for that one new subtree only

Information Gain

Once we split our feature into its following examples, the **entropy reduces to the sum of entropy of the subsets**

$$\text{remainder}(S, x_i) = \sum_{j=1}^{C_i} \frac{|S_j|}{|S|} H(S_j)$$

→ Remainder(S, x_i) – S is the example set (of the feature), x_i is the chosen feature
→ |Sj|/|S| is probability of number of examples in j / total number of examples in sub tree

$$\text{IG}(S, x_i) = H(S) - \text{remainder}(S, x_i)$$

→Remember this subtree/tree has a initial entropy, then after we do the splits, we can calculate the Remainder(S, x_i), and calc IG
→ **IG is ALWAYS positive, so entropy of subsets cannot be higher than original. Min-value = 0**

Cost Functions / Loss Functions

- How to quantify if a hypothesis is close to a target? i.e. $h_\theta \approx f$
→ We use cost functions i.e L(h_θ, f), or J(h_θ, f)
→ Squared Error $(h_\theta(x) - f(x))^2$,
Binary Error $[h_\theta(x) \neq f(x)]$
→ Squared Error for regression functions, binary error for classification (yes or no)
→ $l(h(x), f(x))$ refers to **pointwise cost**, so Overall cost L(h_θ, f) = average of pointwise cost $l(h(x), f(x))$

Training cost:

$$L_{train} = \frac{1}{m}\sum_{j=1}^{m} l(h_\theta(x^{(j)}), f(x^{(j)}))$$

L_{train} is the average of pointwise cost of hypothesis theta. **Average because we don't want to penalize a larger dataset with a higher training cost**.

Test cost:

$$L_{test} = \mathbb{E}_x[l(h_\theta(x), f(x))]$$

→Test cost is way more important than training cost, that's what we are interested in! So we think about the expectation of the point wise loss. Why expectation? Because we don't even know the real answer, since testing is where we are applying our model on data we don't know about, and the test loss would be the expectation of the loss across all the entire space of all possible x
→ Where does our cost function go?

Choosing our cost functions

- Types of costs: False positive (accept) or false negative (reject). How should we penalize for each type? We need to determine which type of cost is more costly in the context of our problem.
→ Sometimes, we cannot use cost functions that fit the task due to lack of information. Then, we use plausible measures:
squared error ≡ Gaussian noise
or convenient measures: **closed form solution**, **convex optimization**
- Where does it go? We use our cost function to compare with our test data to continually tell our learning algo to improve on the errors that it is making on the way, optimizing to reduce loss to a minimum

Ensembles

- Combining various models, unite them, take the result output from the work of more than 1 model
- Several ways of meta-learning: Winner takes all, equal voting (majority wins), weighted vote, and based on condition (given outcome of test, decide which learner to use)
- Key: They should be **diverse;** have different types of biases to look at a problem from different perspectives

Conditional Ensemble – A decision tree

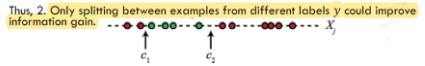
→ DTs are basically conditional ensembles
→ Decision Stump – deciding the output of a label based just on one feature. We decide the threshold and direction, then for the value of that feature of the test data, we decide that it is of x classification

$$h_{s,i,\theta}(x) = s \cdot \text{sign}(x_i - \theta)$$

Where (feature i, threshold θ, direction s)
Optimizing for decision stumps are cheap (i.e. finding the feature and threshold that maximises IG): $O(n \cdot \log m)$ time
→ There are n features and m examples. First, we do binary search across m examples to find optimal split threshold θ. We do this for n features to find the find that maximises reduction in entropy
→ Weak learner, since it allows efficient minimisation of L_{train}, but might be too work to work by itself

Discretizing

- Discretize continuous features. How to choose?



Split at points where the labels are different across 2 examples. That can increase info gain

Pruning

Why prune? If our tree is not pruned, it does not generalize well. Works very well for training data but not the test data. **Little inductive bias**.
Possible methods:
→ Don't let leaves get too small, leaves must have at least i training data
→ Don't let tree grow too deep, cannot have more than j levels of tests
→ Validate performance using part of training data, remove node and replace by mode of labels y if pruned tree performs better than original

DT Regression

→ Instead of mode, use mean
- Test in a leaf predicts the mean of its training instances
→ Performance of uniform blend hypothesis H(x) will be no worse than the average performance over $h_1, h_2, \ldots, h_T$ (proven in tutorial)

Pros and Cons of DT

+ Interpretable – Intuitive for data exploration
+ Efficient, both in training (greedy search, IG) and testing
+ Discards irrelevant features through use of IG
- Instability: Susceptible to small fluctuations, high variance
- Hard decision boundaries, no probabilistic interpretation of boundaries
- Axis perpendicular decisions: Doesn't capture interaction between features (feature & bug)

Things to take note for DTs:

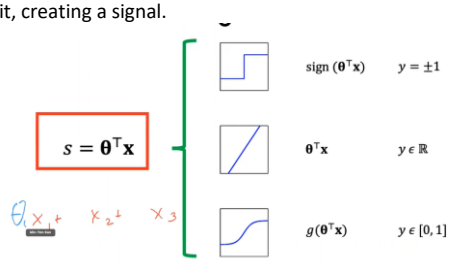
- Features should have quantifiable values to make the explainability less ambiguous
- DTs don't perform well with missing values, replacing with most common value skews dataset, dropping the values reduces size of training set. Replacing with common value skews IG values which affect DTs greatly
- Temporal factors not taken in account in DTs. Maybe a period of bad economy → more rejected loan requests, it will affect DTs greatly when economy improves. Could perhaps remove rows from that time first, but might lead to info loss for cases which would have been rejected when economy was better anyway

Curse of Dimensionality

- In high dimensions, the number of points needed to keep the same density grows exponentially → distance between 2 arbitrary points become similar/almost same distance away, making them hard to distinguish.
→ Learners like KNN that depend on distance breaks down in high dimensions!
→ Euclidean distance is less meaningful

Logistic Classifiers and Logistic Regression

→ When we have a linear model, we take the feature Xs will have a parameter associated with it, creating a signal.



→ $s = \theta^T x$, taking the knobs associated with each feature and multiplying them, then taking the sum. That's the signal
→ we can do sign(s), the signal itself or a g(s), where g is a logistic regression function.
Different ways of coming up with a classification

Formula for linear regression:

$h(\mathbf{x}): \sum_{i=1}^{n} \theta_i x_i > 0$ $\quad \boldsymbol{\theta}^\top \mathbf{X}$ or $\mathbf{X}\boldsymbol{\theta}$?

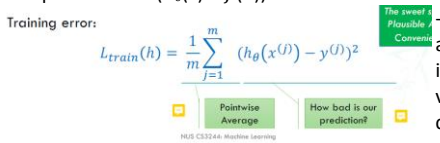
- $\boldsymbol{\theta}^\top \mathbf{x}$ – multiply all knobs with one data point
- $\mathbf{X}\boldsymbol{\theta}$ – multiply entire matrix X, for each data point multiply by the respective knobs

Feature Engineering

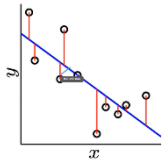
→ Extract useful information from raw info.
→ E.g. for images, extract info such as intensity and symmetry (8 / 3 is symmetric…)
- Transformation of raw data into 'new' features

Linear Regression

- Cost function to approximate how well $h_\theta(\mathbf{x}) = \theta^\top \mathbf{x}$ approximate $f(\mathbf{x})$?
→ Squared Error. $(h_\theta(\mathbf{x}) - f(x))^2$



Important: The lines are parallel to the y axis, because they are measuring the amount of error off the target function (we are looking at the output values, y)

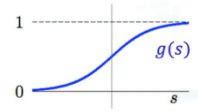


Logistic Regression (sigmoid function)

→ 'Soft' threshold. Linear classification (1 or 0) is has a 'hard' output
→ Logistic function g
→ exp is basically natural log, e

$$g(s) = \frac{exp^s}{1 + exp^s}$$

→ If s = 0, then $exp^0 = 1$, (1 / 1 + 1) = 0.5
→ If s >>>>> 0, exp^s goes towards infinity, so g(s) goes towards 1, i.e. bounded by 1
→ If s <<<<< 0 (negative), exp^s goes towards 0, 0 / 1 = 0, bounded by 0
- g(s) is interpreted as a **probability**. Probability of heart attacks for e.g., with input of cholesterol level, age, weight etc
→ Can be used as linear classifier by setting a cutoff value to the output of sigmoid function



Graph on the left is an example of a sigmoid function

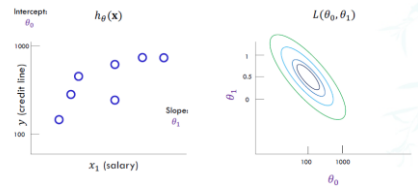
This is the case, even though labels are binary! We are simply seeing the occurrence of an event and trying to infer a probability.

Cost Function for Logistic Regression

$$L_{train}(\boldsymbol{\theta}) = \frac{1}{m}\sum_{j=1}^{m} ln(1 + exp^{-y^{(j)}\boldsymbol{\theta}^\top \mathbf{x}^{(j)}})$$

- $\theta^\top \mathbf{x}$ is the signal, y is the label. If both are very positive or very negative, the exp expression returns a very small number, i.e. a very small cost. Then ln(1 + exp..) is a very small number. (ln(1) gives 0 btw)

Gradient Descent



- For a simple equation that has 2 variables, an x and intercept, we can tweak the slope and intercept, which can give a cost function that varies like the graph on the right. The idea of the circles are that combination of θ_0 and θ_1 values give equally bad cost function.
→ Idea of this is to show that the graph of cost function is like a 'surface' where a minimum point can be found
→ Take 'steps' to minimize loss function, Ltrain (aka in-sample error (Ein), risk minimisation)

Fixed step size η : $\theta(1) = \theta(0) + \eta v$

→ Next value of parameter = original + step size n, v refers to direction (since it's a vector)
- How to decide where to step next? Using the formula for direction v.
→ Taking iterative steps: Batch gradient descent

Formula for the direction v

$$\Delta L_{train} = L_{train}(\theta(t+1)) - L_{train}(\theta(t))$$
$$= L_{train}(\theta(t) + \eta v) - L_{train}(\theta(t))$$
Taylor approximation
$$= \eta \nabla L_{train}(\theta(t))^\top v + O(\eta^2)$$
$$\geq -\eta \|\nabla L_{train}(\theta(t))\|$$

Create v to be a unit vector:

$$v = -\frac{\nabla L_{train}(\theta(t))}{\|\nabla L_{train}(\theta(t))\|}$$

- The idea is that the change is the loss is given by the value at the new step – value at original, and then we use taylor approximation, which approximates a function with an infinite series of terms, go in direction opposite of gradient to maximise reduction of loss (??)
- ∇ (nabla) means gradient
- Time taken in training by GD depends on how many steps it will take to reach minima / threshold

Goldilocks step size

→ Variable step size. When its obvious which way is the right direction, we take large steps.
When the slope becomes gentler and it is less obvious which way is right, we take small steps
→ Varying step size also known as learning rate.
How do we vary? We use the magnitude of the gradient as a way of scaling!
Originally, we have fixed step size

$$\Delta\theta = \eta v$$
$$= -\eta \frac{L_{train}(\theta(t))}{\|L_{train}(\theta(t))\|}$$

$||L_{\text{train}}(\theta(t))||$ == gradient of L_{train} curve. So we multiply by that, and we get

$$\Delta\theta = -\alpha L_{train}(\theta(t))$$

Where α is the learning rate. So now we can tune this learning rate to see how much we want to vary out stepsize. Negative because we are moving in opposite direction of gradient to go to a point less.

Logistic regression algorithm

1. Initialize the weights at $t = 0$ to $\theta(0)$
2. Do
3. Compute the gradient

$$\nabla(t) = \nabla L_{train}(\theta(t)) = -\frac{1}{m}\sum_{j=1}^{m} \frac{y^{(j)}x^{(j)}}{1 + e^{y^{(j)}\theta(t)^\top x^{(j)}}}$$

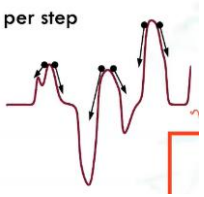
4. // Move in the direction $v(t) = -\nabla(t)$
Update the weights $\theta(t+1) = \theta(t) - \alpha \nabla L_{train}$
5. Continue to next iteration, until it is time to stop
6. Return the final weights θ^*

→ $\theta(0)$ just means that it's the initial weights, not that the weights start as 0!!
→ Calculate the loss
→ Update the weights until its time to return the final weights with almost least cost!
How do we know when to stop?
Criteria:
1. error change is small and/or;
2. error is small;
3. maximum number of iterations is reached.

Stochastic Gradient Descent

In normal gradient descent, in order to minimize training loss, we need to calculate the gradient of the loss function, which is calculated by using the entire training set. What if we only used a part of the training set instead, to increase performance of calculating this gradient?

We take just 1 random point in SGD. We pick a random x* and y*, and calculate its loss. The direction to move down the cost function based off that one data point might be different from the entire batch, but if we do it enough times across each data point, we move in the avg direction, which is correct!



+ Cheaper computation, takes 1/m per step of batch gradient descent.
+ Stochastic: Helps escape local minima
+ Simple
Typically, we do neither batch nor stochastic gradient descent, but mini batch gradient descent, where the original data set is split into small batches, where GD will be performed

Normal Equation of Linear Regression

→ We have a closed form solution for linear regression

$$L_{train} = \frac{1}{m}\sum_{j=1}^{m} (\boldsymbol{\theta}^\top \mathbf{x}^{(j)} - y^{(j)})^2$$
$$= \frac{1}{m}\|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|^2$$

An alternative expression of L_{train}

$$L_{train} = \frac{1}{m}\|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|^2$$
$$= (\mathbf{X}\boldsymbol{\theta} - \mathbf{y})^\top (\mathbf{X}\boldsymbol{\theta} - \mathbf{y})$$
$$= (\mathbf{X}\boldsymbol{\theta})^\top \mathbf{X}\boldsymbol{\theta} - (\mathbf{X}\boldsymbol{\theta})^\top \mathbf{y}$$
$$- \mathbf{y}^\top \mathbf{X}\boldsymbol{\theta} + \mathbf{y}^\top \mathbf{y}$$
$$= \boldsymbol{\theta}^\top \mathbf{X}^\top \mathbf{X}\boldsymbol{\theta} - 2(\mathbf{X}\boldsymbol{\theta})^\top \mathbf{y} + \mathbf{y}^\top \mathbf{y}$$

1. Equation for Ltrain
2. Expand it out, factorize, get the equation above.
3. We differentiate the final equation above with θ.

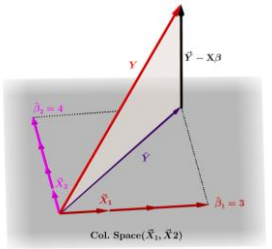
Then solve for $\boldsymbol{\theta}$:

$$\frac{\partial L_{train}}{\partial \boldsymbol{\theta}} \equiv \mathbf{X}^\top \mathbf{X}\boldsymbol{\theta} - \mathbf{X}^\top \mathbf{y} = \bar{0}$$ ← Column of zeroes
$$\mathbf{X}^\top \mathbf{X}\boldsymbol{\theta} = \mathbf{X}^\top \mathbf{y}$$
$$\boldsymbol{\theta} = ((\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top)\, \mathbf{y}$$

- One step learning to get the optimum θ!
- Above is called pseudo inverse.

The Hat Matrix H

→ Projection matrix that expresses the values of the observations in the independent variable y, in teams of linear combinations of the column vectors of the model matrix, X, which contains the observations for each multiple variables youre regressing on



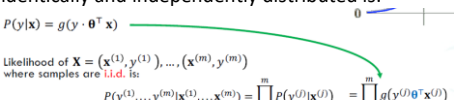
Summary

2 methods for training: Gradient descent, normal equation.
GD – works well, even when n is large, works even if $X^\top X$ is non invertible
Normal Eqn – don't need to choose α, don't need to iterate, but need to compute $(X^\top X)^{-1}$, which is an $O(n^3)$ operation

Logistic Regression Cost Function

$$P(y|\mathbf{x}) = \begin{cases} h_\theta(\mathbf{x}) & \text{for } y = +1; \\ 1 - h_\theta(\mathbf{x}) & \text{for } y = -1. \end{cases}$$

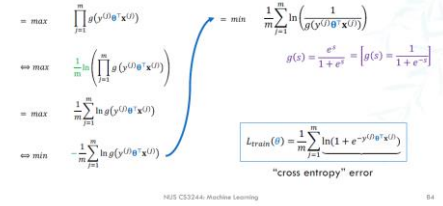
→ Given an x, we calculate the probability of P(y|x) for positive y using $h_\theta(x)$ and use $1 - h_\theta(x)$ to calculate probability of a negative y. (y can be something like how likely a person gets heart disease given all the features x)
→ Likelihood of X, where samples are i.i.d – identically and independently distributed is:



→ Basically, the probability of all the outputs from all the given inputs in a dataset, is just the product of each individual probability since they are i.i.d. We express this as a function of g

We want to maximize the correctness given the data, i.e. minimize our loss function.

Maximizing the likelihood ≡
Minimizing cross entropy



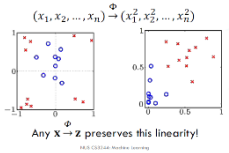
→ So now we want to maximize g
→ We scale it down using log. We still get the same relative ordering so its legal

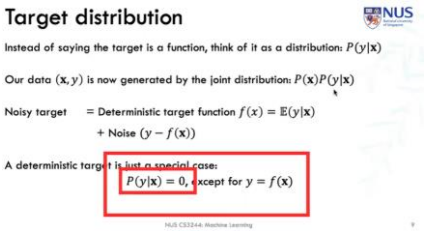
→ We realize that wrapping log in a product is basically summing the powers
→ I want it to look like the cost function, so I add a negative, now my max becomes min
→ Sub the purple g(s) on the right into the eqn to the cross entropy error eqn

$$L_{train}(\theta) = \frac{1}{m}\sum_{j=1}^{m} \underbrace{\ln(1 + e^{-y^{(j)}\theta^{\top}\mathbf{x}^{(j)}})}_{\text{“cross entropy” error}}$$

→ This is a cost function for logistic regression!
→ if same classification, e(high negative) gives small number, so loss is low, else if different, loss is high

Noisy Targets
→ Think of noisy targets as a distribution of x's that give rise to an output y, with some noise in the function.

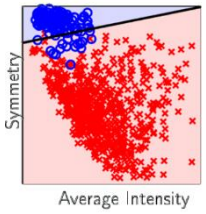




→ Deterministic target is a special case because the probability of an input leading to some output is ALWAYS 0, except at y, so no distribution
→ We see noisy targets in logistic regression a lot!

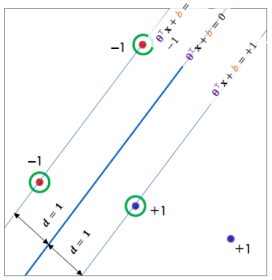
Support Vector Machine SVM
Linear regression alone does not always set the best weights for classification. Its goal is to reduce squared loss, and that might not always work out to the best line of classification

In this case, the classification line is sacrificing too much of the blue just to reduce squared loss, which gives a boundary that has too little room for blue



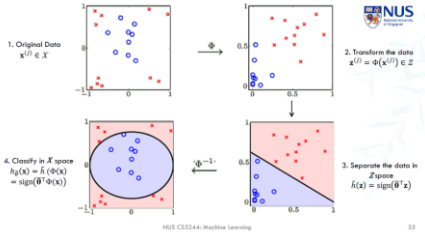
→ We want to construct a line that gives us the biggest margins on both side
→ Handles noisy data well and is good for inputs that turn out to be noisy (noisy inputs not noisy targets!!)

→ Create a hyperplane: $h(\mathbf{x}) = \boldsymbol{\theta}^{\top}\mathbf{x} + b$
→ $\boldsymbol{\theta}$ dictates the orientation of plane (gradient), b dictates offset(bias)
→ Define a distance d to the optimal plane as 1 (unit distance), and try to identify the points that sit at the 2 sides of the support vectors, and define the perpendicular distance from the point to the hyperplane as 1
→ Only a subset of the dataset will determine our unique h(x), known as the support vectors!



Inductive bias of SVM? That we have support vectors and that classes will be separated by margins (i.e. linearly separable)

Non-linear mapping
→ In some models, it might not be meaningful to classify points using a line, but using some non linear separator.
→ Transform the data. Once we do that, we can perform a linear classification on the points, and then transform it back using the inverse of Φ (phi) to get the new non-linear classifier.



→ Using the transformation function, we get a new set of inputs z.

$$\mathbf{x} = (x_1, x_2, \ldots, x_n) \xrightarrow{\Phi} \mathbf{z} = (z_0, z_1, \ldots, z_{\tilde{n}})$$

Note: ñ (from z) may not be equal to n!!
→ Take the entire data matrix and transform it with Φ, and transform all the labels with Φ as well.

→ We didn't have any weights in the X space, but then we will have weights in the z space.

$\boldsymbol{\theta}$? No weights in $\mathcal{X}$ $\tilde{\boldsymbol{\theta}} = (\theta_1, \theta_2, \ldots, \theta_{\tilde{n}})$

Final Eqn:

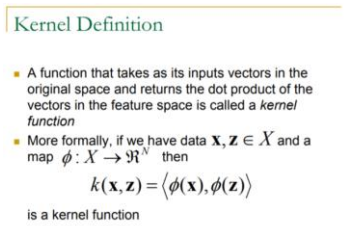
$$h_{\tilde{\theta}}(\mathbf{x}) = \text{sign}(\tilde{\boldsymbol{\theta}}^{\top}\mathbf{z}) = \text{sign}(\tilde{\boldsymbol{\theta}}^{\top}\Phi(\mathbf{x}))$$

→ Support vectors would live in Z space for these transformations
→ They would be maintained in the X space, in a curved fashion, but **the points in the X space closest to the support vectors may not be the ones that were closest in the Z space!!**

Great generalization, since in the model, # of parameters ∝ # of support vectors.

→ Inductive bias is totally dependent on the support vectors, and not to do with the weights of a data point, and the support vectors will dictate the weights (or theta)!

Kernels
→ A function that returns a distance (similarity) measure of two instances.
→ Algorithms like KNN and SVM look at some form of distance to determine the similarity of 2 points. When we meet a family of algorithms like this, we can replace it with a kernel function!
→ Why is it called a "trick"? Originally we have a set of data that we might need to transform to make it linearly classifiable. However, we could instead use the kernel trick, which is some form of modified dot product, to return the distance of the vectors in the higher dimensional space



Soft Margin SVM
- Previously we assumed that our data points were linearly separable and that there was a margin between them. However in soft margin SVM, we allow some errors within our margins of our support vectors

→ We calculate a margin violation:
Margin violation: $y^{(*)}\left(\boldsymbol{\theta}^{\top}\mathbf{x}^{(*)} + b\right) \geq 1$ fails
→ Even if youre on the same side of the highway, if the margin is not >1, it is also considered a violation

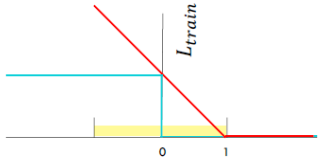
$y^{(*)}\left(\boldsymbol{\theta}^{\top}\mathbf{x}^{(*)} + b\right) \geq 1 - \xi^{(*)}$ where $\xi^{(*)} \geq 0$
Slack variable: Soft error on $(x^{(*)}, y^{(*)})$
Total violation: $\sum_{j=1}^{m} \xi^{(j)}$

→ ξ(*) refers to the arbitrary point that is in violation, and we add that violation as a slack variable. We can count the total violations from this

SVM loss function
→ Hinge loss

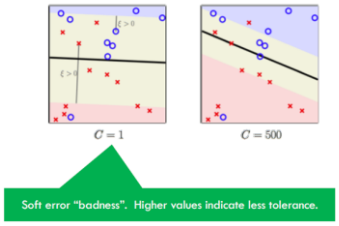
$$L_{train}(\boldsymbol{\theta}) = \sum_{j=1}^{m} max\left(0, 1 - y^{(j)}\left(\boldsymbol{\theta}^{\top}\mathbf{x}^{(j)} + b\right)\right)$$

Positive classifications lead to a number smaller than 0, since we have 1 – (large number). So no error, but if the positive classification < 1, then we incur a tiny error (this means that data point was in the margin)

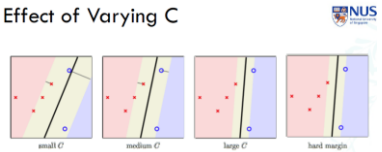


→ Red = hinge loss. When Ltrain > 1, loss = 0, when Ltrain < 1, the loss grows linearly
→ Green = loss in hard margin I think????

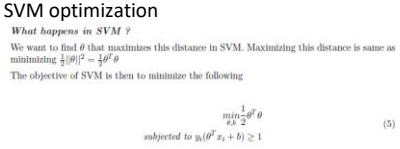
Soft Error C parameter
→ A parameter to define how much we favor correct classifications vs highway size.



→ The larger the C, the harder the SVM

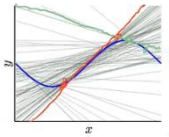


SVM optimization



→ Where y = $\theta^{\top}$ + b

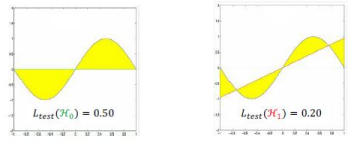
Bias / Variance
→ There is a tradeoff between bias and variance
Bias: The difference between the average prediction and the true value
Variance: The variability of the model prediction for given data, i.e. the spread of our data



Consider 2 models for the sine function:

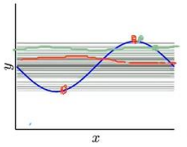
Approximation – $\mathcal{H}_0$ versus $\mathcal{H}_1$

Use the full power of the model

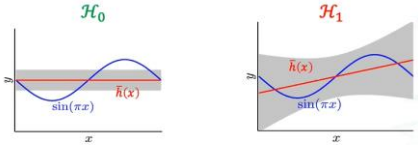


where — and — are the $\bar{h}(x)$ of each $\mathcal{H}$.

→ With these 2 learners above, where $H_0 = \theta_0$ and $H_1 = \theta_1 x + \theta_0$, in an ideal case where we get to sample many points, this are the average hypotheses ***h̄(x)* bar** that we would love to get. However, in ML, we only get one chance to sample/train our model!

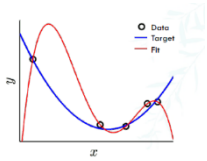


→ As seen in the models, given 2 data points, the variabilities of the possible H0 and H1 we can get are shown. While we might have a higher potential to do a little better for H1, if we get 2 datapoints then there is a very high chance we perform badly
→ Even though for approximation, H1 performs much better than H0, we are NOT doing approximation but learning in ML, in the lens of a particular dataset, so our models have a higher sensitivity to which points were drawn from the underlying target function.



Bias = 0.50 Var = ? 0.25
Bias = 0.21 Var = ? 1.69

→ In ML, its different in stats, we are merely learning from the dataset (a subset of the entire realm of possibilities).
→ Match the 'model complexity' to the



data resources, not to the target complexity. It really depends on how clean / how many data points you have before you determine the complexity of the model you are going for

Learning Curves

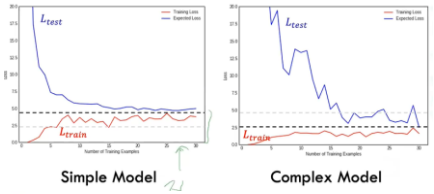
- Look at the tradeoffs between training loss and testing loss as we vary the size of the dataset

→ Theoretical

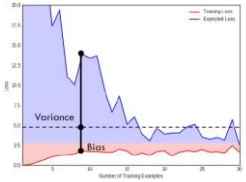
→ We are interested in the expected test cost and training cost, across all the possible permutations of sets of data of size x, given that we have all the data related to our problem

Expected test cost $\mathbb{E}_{\mathcal{D}}[L_{test}(f_{\mathcal{D}})]$

Expected training cost $\mathbb{E}_{\mathcal{D}}[L_{train}(f_{\mathcal{D}})]$



Bias–Variance on a Curve



→ The graphs of the expected test/training costs. In a simple model, we can expect the overall loss (both) to be higher, because we don't have a very good way of representing the target function. In the long run (if we have more training examples), then we can see that the expected cost goes towards the dotted line, because we cannot represent it perfectly with a simple model

→ Complex model has a lower expected overall loss in the long run. Can fit the data better, so it gets closer and closer to 0. For complex model, we have loss that it quite jagged, i.e. there is more variability in the expected loss, because there are more parameters in play

→ Why do we have lower training loss when there are less examples? Because when we have less points, we have the tendency to fit exactly, so the training loss is low (may not be a good thing). So more data → increasing training loss, but testing error goes down, because we get a model that generalizes better

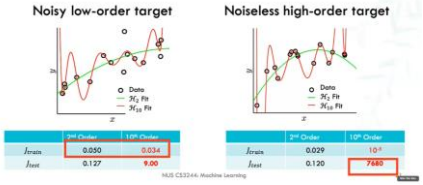
Overfitting

Take the instance we have 5 points like this, where they are slightly noisy and the target function is a 4th order polynomial fit, and we fit the data perfectly.

We might get a very low training loss, but when we look at the actual polynomial, the difference between our target function and hypotheses is huge! Our hypothesis is terrible.

→ Fitting the data more than is warranted – overfitting

→ Fitting the noise causes our model to really fluctuate, i.e. it is not just useless, but harmful, especially if we try to fit our training data too well. Our model cannot differentiate between signal (good) vs noise (bad)



→ Given 15 data points, would you rather pick a 10th order target + noise or 50th order target noiseless?

→ On the left, given noisy low order target, because our hypotheses is trying to fit so well to the data, we get this curvy hypothesis that is trying to fit to every single data point, keep in mind these points are noisy so overfitting can cause testing loss to be very high

→ In the example on the right, where we have a 50th order polynomial, we still can fit the training data perfectly, getting very low training loss, but that does not matter, if you overfit your data, it will not generalize well → high test loss

Approximation vs Generalization

What we want → Small test loss, so we will need good approximation of f (target) on unseen data

Tradeoff: More complex H → better chance at approximating f

Less complex H → Better chance of generalizing on test data

We need to find a balance/best H to balance these tradeoffs

Occam's Razor (ML Context)

→ Simpler is better (in generalizing on data you have not seen before)

Various Models w.r.t Variance

KNN - Model free: Data completely dictates the model

Higher k lowers the dependence of the model on a particular data point, makes model more robust and lowers variance

Decision Trees - Pruning discards detailed tests that may use criteria nonessential for classification in test data, i.e. criteria that is actually useless w.r.t. to the set of all data

Linear Models – Each additional parameter θ_i adds a degree of freedom in the model, can approximate better but higher variance

SVM – Deals with variance better, because its main focus in creating the vectors are based on the support vectors, if the other points are off, it does not affect the support vectors' placement at all

Ensembles – Different hypothesis have different ways at looking at the input space – diversity: Different viewpoints, different tribes, do not solely depend on one tribe with a certain inductive bias. Ensembling diverse h generalizes better!

Bias & Variance Bias Variance Tradeoff

We want to know how well our Ltest decomposes into 2 parts:

1. How well can H approximate f overall (bias)
2. How well can we zoom in on a good H_θ in the set of all hypotheses, i.e. How to find a good hypothesis

We will look at this through squared error

Terminology:

$\mathbb{E}x[z(x)] \equiv$ Expected value of $z()$, given the distribution of values of x.

$h_{\mathcal{D}} \equiv$ Hypothesis of learner when learning from Dataset $\mathcal{D}$.

$\mathbb{E}_{\mathcal{D}}[z()] \equiv$ Expected value of $z()$, given distribution of possible Datasets $\mathcal{D}$.

$L_{test}(h_{\mathcal{D}}) = \mathbb{E}_x[(h_{\mathcal{D}}(x) - f(x))^2]$ — $h(x)$ depends on the specific dataset $\mathcal{D}$.

$\mathbb{E}_{\mathcal{D}}[L_{test}(h_{\mathcal{D}})]$ — Generalizing over all $\mathcal{D}$ with same m

$= \mathbb{E}_{\mathcal{D}}[\mathbb{E}_x[(h_{\mathcal{D}}(x) - f(x))^2]]$

$= \mathbb{E}_x[\mathbb{E}_{\mathcal{D}}[(h_{\mathcal{D}}(x) - f(x))^2]]$ — Swap ok, as integrand is strictly non-negative

Focus on $\mathbb{E}_{\mathcal{D}}[(h_{\mathcal{D}}(x) - f(x))^2]$, get the outer term $\mathbb{E}_x[\dots]$ later.

→ Now we have this expression above, we are interested in the average hypothesis. The average hypothesis is basically the theoretical scenario where we have the set of all possible data points, we draw m number of points from it to create a hypothesis h, then we average them to get an avg hypothesis.

To evaluate $\mathbb{E}_{\mathcal{D}}[(h_{\mathcal{D}}(x) - f(x))^2]$, we define the 'average' hypothesis $\bar{h}(x) = \mathbb{E}_{\mathcal{D}}[h_{\mathcal{D}}(x)]$

Imagine many, many data sets $\mathcal{D}_1, \mathcal{D}_2, \dots, \mathcal{D}_K$ drawn:

$$\bar{h}(x) \approx \frac{1}{K}\sum_{k=1}^{K} h_{\mathcal{D}}(x)$$

→ Kind of like ensembling, arguably the 'best' hypothesis I can get based on x number of hypotheses of size m with my data set I have

Using $\bar{h}(x)$

$\mathbb{E}_{\mathcal{D}}[(h_{\mathcal{D}}(x) - f(x))^2] =$

$= \mathbb{E}_{\mathcal{D}}\left[(h_{\mathcal{D}}(x) - \bar{h}(x) + \bar{h}(x) - f(x))^2\right]$ — Add $-\bar{h}(x) + \bar{h}(x)$. Associate first two and second two terms.

$= \mathbb{E}_{\mathcal{D}}\left[\left(h_{\mathcal{D}}(x) - \bar{h}(x)\right)^2 + \left(\bar{h}(x) - f(x)\right)^2 + 2\underbrace{\left(h_{\mathcal{D}}(x) - \bar{h}(x)\right)}_{\text{const w.r.t. } \mathcal{D}}\left(\bar{h}(x) - f(x)\right)\right]$ — Expand out, cross terms drop as first part of term is 0.

$= \mathbb{E}_{\mathcal{D}}\left[\left(h_{\mathcal{D}}(x) - \bar{h}(x)\right)^2\right] + \left(\bar{h}(x) - f(x)\right)^2$ — 2nd term is constant with respect to $\mathcal{D}$

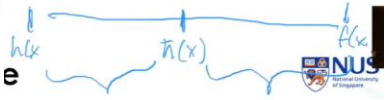
→ Expand out the entire term in the second step, the term that was grouped to be 0, is because we defined the average hypotheses, h-bar, to be the expected value of a hypothesis across the entire dataset, so the 2 terms are equal, so the entire term at the end is gone

→ In the third step, the second term (where my first yellow bubble is) is not related to D so can take out of the entire bracket.

→ At the very end, the second term with f(x) is bias, first term is variance

→ Second term: Bias is the measure of how well our hypothesis can approximate the target function (how powerful your hypothesis is), and h-bar is the best we can do, h-bar is not about a particular dataset, and h-bar also measures whats the best we can do for our model, since its taking the avg across all possible hypothesis, not solely based on any single one of them

→ First term: Variance. Variance is a measure that is specific to a single hypothesis, and the first term is pretty much the formula for variance

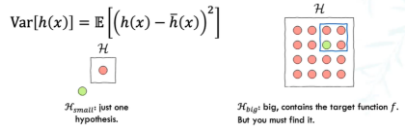


→ Therefore, the expected loss of a hypothesis of a dataset is basically bias + var. The distance from h(x) to f(x) is our loss

The tradeoff

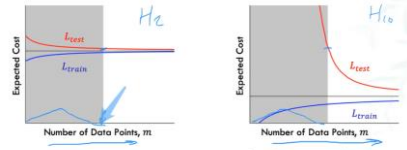
$\text{Bias}[h(x)] = |\bar{h}(x) - f(x)|$

$\text{Var}[h(x)] = \mathbb{E}\left[\left(h(x) - \bar{h}(x)\right)^2\right]$



→ H_{big} basically refers to a hypothesis where I have a very big polynomial, with a big polynomial, it has a higher chance of containing f, but I have so many parameters that it may not be so easy to find the target function. And also, the variability increases a lot, compared to a small H

Overfit Measure

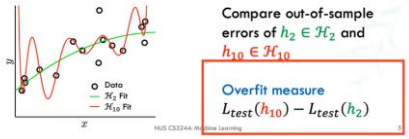


→ Up to the point of the edge of the grey region, using H10 has a greater loss than H2, because we simply to not have enough points to model a polynomial that big, and we risk overfitting. If we have more m, which allows a better/more efficient fit of the data (judging w.r.t. L_{test})

→ For H2, we can also see the expected cost (dotted line) is much higher. Because it's a simpler polynomial that cannot fit the target function as well

We fit the data set $(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_m, y_m)$ using our 2 models:

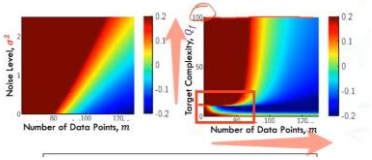
$\mathcal{H}_2$: 2nd order polynomials $\quad$ $\mathcal{H}_{10}$: 10th order polynomials



→ For the overfit measure, we say that H_{10} overfits more than H_2 (which is why it has a higher test loss right)

(Slide 52, bias-variance)

Overfit measure: $L_{test}(h_{10}) - L_{test}(h_2)$



→ Look at the scale to the right of each graphs. If we are at the green area, it means that both H2

and H10 have comparable performances, If we go towards red it means we overfit, and using H2 would have been better than H10, if it goes towards blue then using H10 was better (because we have better data resources)

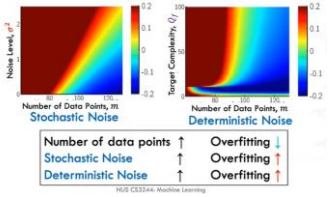
→ Left graph is simply saying with more and more data points, I have less overfitting and can probably use H10 as a better fit. This depends on the amount of noise as well

→ With the graph on the right, the y axis refers to the complexity of the target function. If our target complexity is extremely high, then if the number of data points that we have are insufficient, then we would always be overfitting, because we simply do not have enough points to model the target function

→ The weird intercept on the second graph is the 10th order polynomial (remember we are using a H10 in this case), and it exists because we might be able to fit to the model nicely much faster if we have enough data points

→ If we are fitting a target of <10th order polynomial, we overfit, when we have too little data points.

→ Left graph is stochastic noise and right graph is deterministic noise (bias)



Role of noise

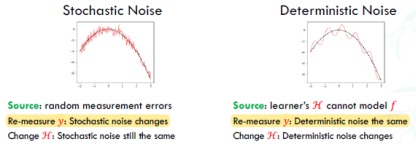
→ The part of y that we cannot model. 2 sources: Stochastic and Deterministic noise

→ Stochastic noise (or irreducible error): Fluctuations because our way of measurement for example, was not exact

→ Right elephant is blur/pixelated

→ Deterministic noise: Our model does not have the capacity to model f, i.e. our model is not powerful enough

- Both are hurtful, but are due to different reasons.



→ note that in reality, we only have 1 dataset and a fixed H

Noisy Targets

$$y = f(x) + \epsilon(x) \qquad \mathbb{E}[\epsilon(x)] = 0$$

→ Let's say we have a noisy target, but the expected value across all is still 0 (there are fluctuations on both sides). We are trying to express y (of the form below) as a expression of bias, variance and stochastic noise

$$\mathbb{E}_{\mathcal{D},\epsilon}[(h_{\mathcal{D}}(x) - y)^2] =$$
$$= \mathbb{E}_{\mathcal{D},\epsilon}[(h_{\mathcal{D}}(x) - f(x) - \epsilon(x))^2] \quad \text{Expand observed } y$$
$$= \mathbb{E}_{\mathcal{D},\epsilon}\left[\left(\underbrace{h_{\mathcal{D}}(x) - \bar{h}(x)}_{①} + \underbrace{\bar{h}(x) - f(x)}_{②} - \underbrace{\epsilon(x)}_{③}\right)^2\right] \quad \text{Associate terms}$$
$$= \mathbb{E}_{\mathcal{D},\epsilon}\left[\left(h_{\mathcal{D}}(x) - \bar{h}(x)\right)^2 + \left(\bar{h}(x) - f(x)\right)^2 + (\epsilon(x))^2 + \cancel{\text{cross terms}}\right] \quad \text{Additional cross terms disappear as } \mathbb{E}(\epsilon(x)) = 0$$

$$\mathbb{E}_{\mathcal{D}}[(h_{\mathcal{D}}(x) - f(x))^2] =$$
$$\underbrace{\mathbb{E}_{\mathcal{D},x}\left[\left(h_{\mathcal{D}}(x) - \bar{h}(x)\right)^2\right]}_{\text{variance}} + \underbrace{\mathbb{E}_x\left(\bar{h}(x) - f(x)\right)^2}_{\text{bias}} + \underbrace{\mathbb{E}_{\epsilon,x}(\epsilon(x))^2}_{\sigma^2}$$

Deterministic Noise — Stochastic Noise

Regularization

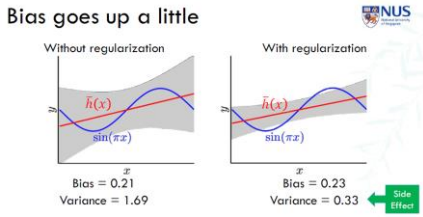
→ Restrains the model, estimates a form of overfit penalty to reduce overfitting

→ A cure for our tendency to fit the noise → improve Ltest

→ Does it by constraining model so that we cant fit noise

→ Side effect: Being unable to fit noise might mean we cannot fit signal f, increasing the bias error because there are now more possible target functions that we can't fit

→ But reduce the probability of overfitting a lot



→ Bias increases, but the variance decreases a lot more

$$\mathcal{H}_{10} = \{h(x) = \theta_0 + \theta_1 x + \cdots + \theta_{10} x^{10}\}$$

$$\mathcal{H}_2 = \{h(x) = \theta_0 + \theta_1 x + \cdots + \theta_{10} x^{10}\} \quad \text{such that } \theta_3 = \theta_4 = \cdots = \theta_{10} = 0$$

A "hard" order constraint that sets some weights to zero.

$$\mathcal{H}_2 \subset \mathcal{H}_{10}$$

→ Idea behind regularization: We know that a model like H_2 is already in H_{10}, i.e. is a subset of H_{10}, because we can just tune the parameters in H_{10} to make it a H_2. We also know that advantages a lower polynomial can bring in reducing variance. Is there a way we can favor taking hypotheses that resemble H_2, while using the H_{10} polynomial?

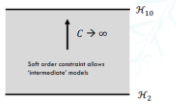
→ Currently, by setting a "H2", we are basically setting a "hard" order constraint that sets the weights to zero. But what we want is something more gradual, a "soft" order constraint

Soft Order Constraint

→ We give a budget. The learner will choose how to use this budget, and the weights of each parameter, after going through some regularization function should be not more than the budget

Re- use loss optimization by giving a budget and let the learning choose. Introduce a regularization function $\Omega(h)$.

$$\Omega(h) \equiv \sum_{q=0}^{Q} \theta_q^2 \leq C$$



$$\mathcal{H}_{10} = \{h(x) = \theta_0 + \theta_1 x + \cdots + \theta_{10} x^{10}\}$$

$$\mathcal{H}_C = \{h(x) = \theta_0 + \theta_1 x + \cdots + \theta_{10} x^{10}\} \quad \text{such that } \sum_{q=0}^{10} \theta_q^2 \leq C$$

N.B. $\mathcal{H}_2$ and C=2 are unrelated!

$$\mathcal{H}_2 = \{h(x) = \theta_0 + \theta_1 x + \cdots + \theta_{10} x^{10}\} \quad \text{such that } \theta_3 = \theta_4 = \cdots = \theta_{10} = 0$$

$\mathcal{H}$ is larger

$\mathcal{H}$ is smaller

→ We can tune how "conservative" our model is by tuning how big of a C we give, a bigger C who mean less regularization and more variance

→ Note: Budget of 2 DOES NOT MEAN H2. The coeffiecients can be added across any of the polynomial coefficients

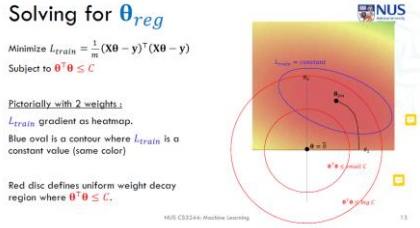
→ Why does limiting this C even makes sense? It is something to do with the model complexity of ML problems, when we make them human-understandable, they typically don't involve a large number of coefficients; mild coefficients are favoured in human curated datasets, which is why regularization generally works well.

- We want to solve for $\boldsymbol{\theta}_{\text{reg}}$ (regularization parameter)

Minimize $L_{train} = \frac{1}{m}(\mathbf{X\theta} - \mathbf{y})^\top(\mathbf{X\theta} - \mathbf{y})$

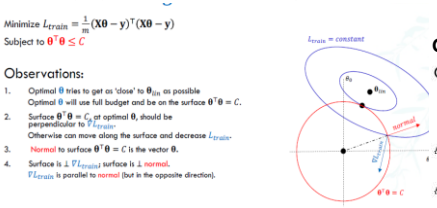
Subject to $\boldsymbol{\theta}^\top\boldsymbol{\theta} \leq C$

Minimize Ltrain ←



→ We have the region where given our budget, the possible Ltrain values we might get, depicted by red circle. Big circle means bigger C (more likely to encompass solution with optimal Ltrain)

→ We have a region of a blue ellipse, that region regions of constant Ltrain. Why isit an ellipse not a circle? In this case, x-axis is $\boldsymbol{\theta}_1$ and y axis is $\boldsymbol{\theta}_0$, varying them affects the loss differently. In this case, varying $\boldsymbol{\theta}_0$ affects the training loss more



→ Choose the point on the disc to get as close as exact solution. Perpendicular = lower loss, because with a budget that is not enough to actually achieve $\boldsymbol{\theta}_{\text{lin}}$, when we are at the point with lowest lost, its perpendicular to -gradient of Ltrain

$$\nabla L_{train}(\boldsymbol{\theta}_{reg}) \propto \boldsymbol{\theta}_{reg} = -\frac{2}{m}\boldsymbol{\theta}_{reg}$$

Constant of proportionality. Chosen for convenience

There is a correspondence: $C \uparrow \quad \lambda \downarrow$

→ We usually express regularization as a constant of expression, lambda, in libraries like sklearn, smaller lambda means bigger budget, i.e. less regularization.

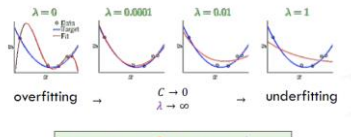
Closed Form Solution of linear regression with regularization:

$$\boldsymbol{\theta}_{reg} = (\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^\top\mathbf{y}$$

→ Similar to closed form solution of linear regression without regularization

Effects of regularization:

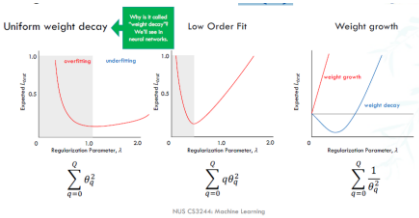
Minimizing $L_{train}(\boldsymbol{\theta}) + \frac{\lambda}{m}\boldsymbol{\theta}^\top\boldsymbol{\theta}$ for different λ's:



Q1: What happens θ to in the limit as $\lambda \to \infty$?

Regularization Variants: $\Omega(h)$

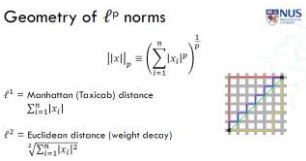
→ Uniform weight decay, low order fit, weight growth



→ We want the minimum points of all the graphs (lowest Ltest). Too little lambda and we overfit, too much we start to underfit.

→ Note: In general, weight growth is not really good. Optimum amount seems to be 0

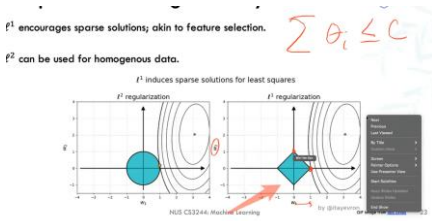
General Solution for Lp norms



Different spaces (shapes) with different Lp:

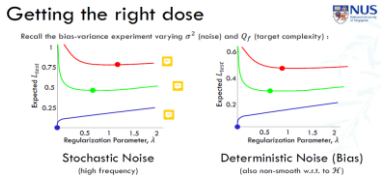
As the value of p decreases, the size of the corresponding space also decreases.

In 3 equally weighted dimensions (e.g., x_1, x_2, x_3):



→ Consider for the right graph (l1), why is the region bounded by our budget, in the shape of the polyhedral.
→ Define x-axis as W1 and y-axis as W2. If we give all our budget to W1, then W2 is 0, and we have the point on the x-axis = 1. As we allocate more to W2, the distribution of weights is proportionate, so we get a linear line from W1 = 1/W2 = 0 to W2 = 1/ W1 = 0.
→ Apply this logic to L2, and we get the circle (not directly proportionate because squaring of terms is involved)
→ This graph helps us to observe what values are being explored when we set our regularization budget given different Lp norms
→ L1 encourages sparse solutions; where some of the features get 'turned off'.

How to decide right amount of lambda?



→ We can see from the blue lines. If our model has no noise, then 0 regularization is best as the expected Ltest is 0. Regularization helps to reduce the variance of our Ltest across different levels of noise
→ No matter what kind of noise, regularization solves the same problems

Is there a Perfect regularization function Ω(*h*)?

→ Occam's razor – Simpler model has higher chance of generalizing
→ Even if we choose a bad function Ω(*h*), we can set the amount lambda (validation)

Regularization Summary

→ We are giving up modelling a subset of H for lower variance error

$$L_{aug}(\boldsymbol{\theta}) = L_{train}(\boldsymbol{\theta}) + \frac{\lambda}{m}\Omega(h)$$

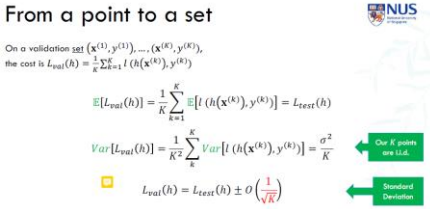
→ New loss = original training loss + regularization (tuned by amount of lambda constraint and size of dataset)
→ Choose lambda using validation

Data Snooping

→ When we normalize our data, we need to split our data to Xtrain, Xtest first, before normalizing. Else we snoop on our data, because we have taken into account our test data's mean and variance with our training data's, and that small amount of contamination lead to a model overfitting!
→ If a data set has affected any step in the learning process, its ability to assess the outcome has been compromised.

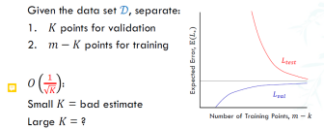
Validation

→ Measure against overfitting by peeking/reality checking at the bottom line. We should try to make the loss on our training data as close to the test data as possible (good measure of appropriate fitting)



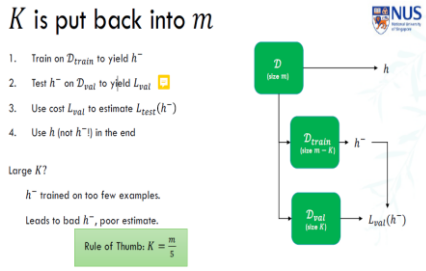
→ For a validation set, validation loss, Lval, is the average of the sum of all pointwise loss (same as usual)
→ Expected validation loss is the average of the expected loss over the K points in the validation set, should be the same as the testing loss
→ Variance – Our k points are all independent points. When we keep measuring variance, the variance reduces by the square. **The more points we have in our validation set, the more our variance reduces, the more stable our estimates of our test costs are**

→ Factor reduction in variance with more K
→ Again, we need to balance between having sufficient amount of points for validation, K, and points to train, m - K
→ Rule of thumb: m / 5 for validation



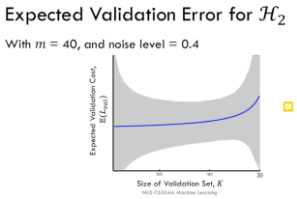
→ Large K leads to bad h due to poor estimate

How to use validation in training our model



→ After all validation, we still use the h, with all the parameters set on the full dataset, and use that as our hypothesis

Expected Validation Error for H



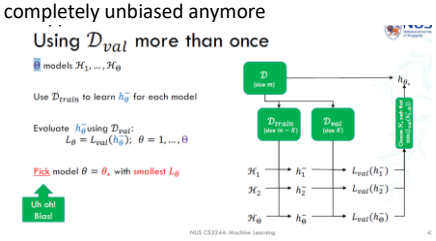
→ Larger K, less m – K, we have less data to fit h-, not going to fit as well, loss increases
→ The shaded areas are our **variance** in loss. So we want to choose the point that minimizes variance, but still have a solid amount of training data such that expected Lval is not affected too much. Too little validation data K is also not good because there isn't enough repetition to lower variance that we expect

Cleanliness of Validation

→ Why is a validation set called that? Because it is used to make learning choices (i.e. choose level of regularization to minimize Lval).
→ If an Ltest affects learning, its no longer a test set but a validation set. A test set gives a reliable estimate of the test cost because it is completely unbiased.

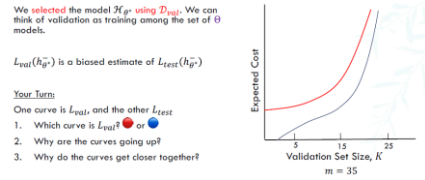
A validation set has an optimistic bias. Why? Because with 2 hypotheses ha and hb, where the average Ltest = 0.5, we are taking the one with lower loss, so expected loss (E(l)) < 0.5, resulting in optimistic bias

→ Take for instance we have a dataset D, and we train Θ models H1, … , HΘ, choose the one that got the best result (i.e. least loss), choose it to optimize over the entire data set and return the best parameters. By choosing/picking, we have introduced bias to our training. It's like 'fitting' our data to the validation set, so its not completely unbiased anymore



Data contamination

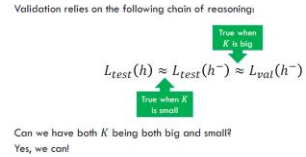
Lval vs Ltest



→ Red – Ltest, Blue – Lval

Cross Validation (K Fold)

- Relationship between high K and loss



→ When we have small K, Ltest(h-) is comparable to Ltest(h) because the data used for both are comparable (since size of K is small)
→ When we have large K, our validation becomes more stable, so validation and test cost become comparable, but because m – K is smaller, Ltest(h) and Ltest(h-) are not as comparable anymore
→ Use cross validation to get best of both

Leave out out cross validation

$m - 1$ points for training and 1 point for validation
(Sounds familiar? It was at the beginning of the pre validation lecture)

$\mathcal{D}_{cv} = \left(\mathbf{x}^{(1)}, y^{(1)}\right), \ldots, \left(\mathbf{x}^{(cv)}, y^{(cv)}\right), \ldots, \left(\mathbf{x}^{(m)}, y^{(m)}\right)$ validation

Final hypothesis learned from $\mathcal{D}_{cv}$ is h_{cv}^-.

$$l_{cv} = l_{val}(h_{cv}^-) = l\left(h_{cv}^-\left(\mathbf{x}^{(cv)}\right), y^{(cv)}\right)$$

Caveat: Hypothesis learned will be highly correlated.
As most points are identical: The 1st hypothesis will use points 2, 3, …, m and the 2nd will use 1, 3, … , m.

Take the scenario where we only remove one point for validation, train models where we don't have that 1 point used for validation, all our models will be highly correlated as most points are identical
→ known as leave-one-out cross validation (LOOCV)
→ Very expensive for large datasets and might have higher variance as well, as test error estimates are highly correlated

K-Fold Cross Validation

→ K non overlapping folds, using one fold as validation the rest as training
→ Use 10 fold or 5 fold
→ The correlation between folds are still high because they are overlapping

→ If cross validation were averaging across independent estimates, then LOOCV should see lower variance, since training sets overlap substantially. But when the training sets are highly correlated, LOOCV is blind to instabilities, that may not be triggered by changing a single point in the training data, makes it highly variable to realization of training set, hence high variance

Summary

→ More data points overfitting less likely to occur, more noise/target complexity overfitting more likely to occur
→ If H is constant, and f complexity increases, deterministic noise increases in general, stochastic not affected, higher tendency to overfit (because our model will fit to our points, not the target function)
→ If H increase, f is fixed, deterministic noise decrease in general, stochastic noise constant, higher tendency to overfit (because we are more likely to fit noisy data points)

Classification Evaluation Metrics

Correctness

→ Classification is correct when prediction y-hat is the same as actual label y

$$\text{Correct} = [\hat{y} = y]$$

where

- $\hat{y} = M(\boldsymbol{x})$ is the predicted value from model M instance $\boldsymbol{x}$
- y is the ground truth value
- $[P] = \begin{cases} 1 & \text{if } P \text{ is true} \\ 0 & \text{otherwise} \end{cases}$ is the Iverson bracket notation for if/else

Accuracy

"Average Correctness" across test dataset with m instances:

$$A = \frac{1}{m}\sum_{j=1}^{m} [\hat{y}_j = y_j]$$

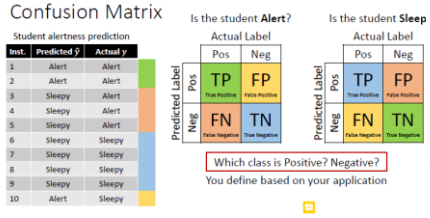
where

- $\hat{y}_j = M(\boldsymbol{x}_j)$ is the predicted value from model M of the jth instance $\boldsymbol{x}$
- y_j is the ground truth value of the jth instance
- $[P] = \begin{cases} 1 & \text{if } P \text{ is true} \\ 0 & \text{otherwise} \end{cases}$ is the Iverson bracket notation for if/else

→ Average of all correctly predicted / instances predicted

Confusion Matrix

→ We explicitly define our what is positive with respect to our confusion matrix.



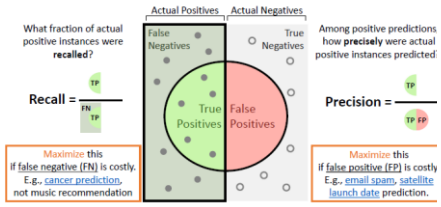
→ Once we have a confusion matrix, we can evaluate which metrics are more important; for e.g. do we want to penalise FPs or FNs more

Cost-Sensitive Evaluation Metrics

Precision vs Recall

→ Recall: TP / TP + FN

→ Precision: TP / TP + FP



→ Maximize Recall if FNs are costly (cancer prediction)

→ Maximize precision if FPs are costly (email spam)

F1 Score (Harmonic Mean)

→ Calculated as some form of balanced metric between precision and recall, measures test accuracy

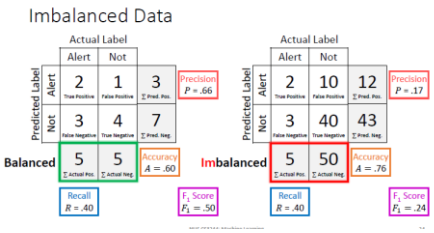
F1 = 2 / (1/P) + (1/R)

→ Much more robust than simple averages (less sensitive to extreme values)

→ Compares the denominators of both P and R

$$F_1 = \left(\frac{P^{-1} + R^{-1}}{2}\right)^{-1} = \frac{2}{\frac{1}{P} + \frac{1}{R}} = \frac{2\,TP}{(TP + FP) + (TP + FN)}$$

Imbalanced Data

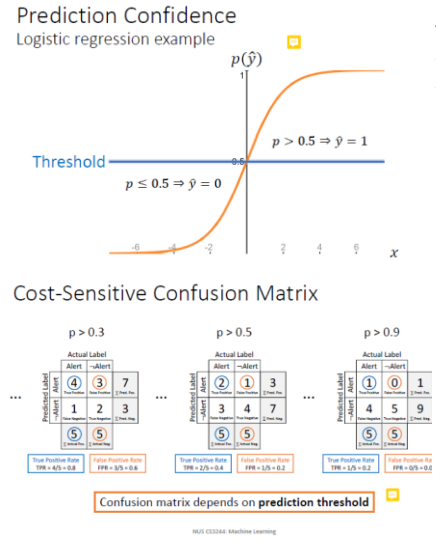


Consider the following confusion matrices, where on the left, the data is balanced (5 +ve, 5 -ve) vs on the right imbalanced (5 +ve, 50-ve)

→ We might have a better accuracy on the right because we have a lot of correctly predicted negatives, but do we actually care about it? (Since P and R are actually really low)

Prediction Confidence

→ How confident the model is in predicting an outcome. Different from probability, because probability is something that should never change even if the model changes



→ By varying the threshold values to determine the prediction, we get confusion matrices of different values.

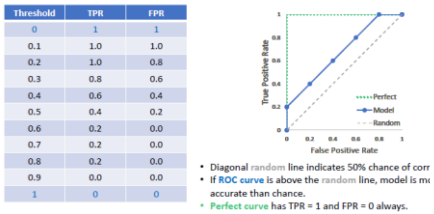
→ In general, the higher the threshold, true positive rate and false positive rate goes down

True Positive Rate	False Positive Rate
TPR = TP / (TP+FN)	FPR = FP / (FP+TN)

→ TPR is calculated w.r.t all the positive labels, FPR calculated w.r.t all negative labels

Receiver Operator Characteristic (ROC) Curve

- The diagonal dotted line indicates 50% chance of correctness by chance.
- If your ROC curve is above the random line, the model is more accurate than chance. Perfect curve has TPR = 1 and FPR = 0 for all thresholds



→ As your threshold increases, the number of predicted positive decreases, so true positive prediction decreases, TPR reduces. The number of false positives also decreases, FPR reduces.

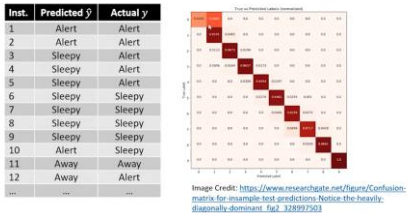
→ AUCROC

- With the ROC curve, integrate it and get a value from 0 to 1, to summarize each individual result in the ROC curve.

– concise metric, enable clearer comparisons. AUC>0.5 means model is better than chance, AUC around 1 means model is really accurate (and really confident)

Multiclass Classification / Confusion Matrix

→ Confusion matrix for each individual class



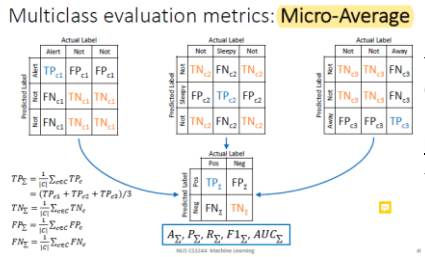
→ We want most of the values to be in the centre diagonal (that means that actual = predicted and is the positive label).

Evaluation Metrics for Multi-class

→ Accuracy: Take the middle diagonal, sum that, divided by sum of all the squares

→ With respect to just one of the labels, we can arrange the prediction vs labels of each data point, then only the one that is labelled TP is the predicted positive = labelled positive.

Micro-Average



→ General Idea: We are calculating each confusion matrix for each label

→ We take each individual values and create and accumulated confusion matrix and average that

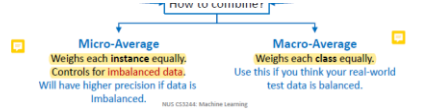
Macro-Average

→ General Idea: We calculate the individual metrics for each confusion matrix, then we average them out respectively for accuracy, precision, recall, F1, AUC.

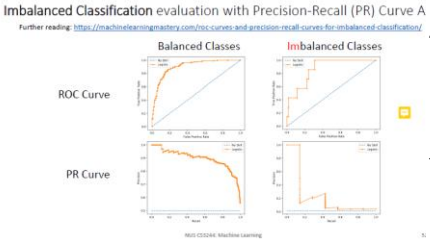
$$A_\Sigma = \frac{1}{|C|}\sum_{c\in C} A_c \qquad P_\Sigma = \frac{1}{|C|}\sum_{c\in C} P_c \qquad F1_\Sigma = \frac{1}{|C|}\sum_{c\in C} F1_c$$

$$R_\Sigma = \frac{1}{|C|}\sum_{c\in C} R_c \qquad AUC_\Sigma = \frac{1}{|C|}\sum_{c\in C} AUC_c$$

Micro vs Macro-Average



- Use micro-average when we need to weight each instance or prediction equally
- Use macro-average when all classes need to be treated equally to evaluate the classifier's overall performance concerning the most freq class labels.



→ Look at the general shape of the 4 graphs

→ Left side are balanced classes, right are imbalanced. Top graph is AUCROC, bottom is precision-recall curve.

→ If the classes are imbalanced, we see that the PR curve is very skewed, because it is predicting one majority class very well, whereas its predicting the rest not as well, resulting in the graph having this weird shape.

→ If this is happening then our model's accuracy might be high because it is over-represented by that class, but that doesn't mean our model is doing that well at all

Regression Evaluation Metrics

- Regression Incorrectness "Difference"

→ Regression is less correct when prediction y is more different from the actual label y

$$\text{Difference} = d(\hat{y}, y)$$

where

- $\hat{y} = M(\boldsymbol{x})$ is the predicted value from model M instance $\boldsymbol{x}$
- y is the ground truth value
- $d(\hat{y}, y)$ is some distance calculated between $\hat{y}$ and y

→ We quantify this difference using some form of distance metric

Difference metrics per instance

1. Simple difference: d = y-hat – y

→ Treats positive and negative difference differently

2. Unsigned difference: d = |y-hat - y|
3. Squared Error (y-hat - y)^2 – penalizes large differences more than absolute error

Average difference metrics for regression

1. Mean Absolute Error (MAE)

$$MAE = \frac{1}{m}\sum_{j=1}^{m} |\hat{y}_j - y_j|$$

2. Mean Squared Error (MSE)

$$MSE = \frac{1}{m}\sum_{j=1}^{m} (\hat{y}_j - y_j)^2$$

3. Root Mean Squared Error (RMSE)

$$RMSE = \sqrt{\frac{1}{m}\sum_{j=1}^{m} (\hat{y}_j - y_j)^2}$$

4. Mean Absolute Percentage Error (MAPE)

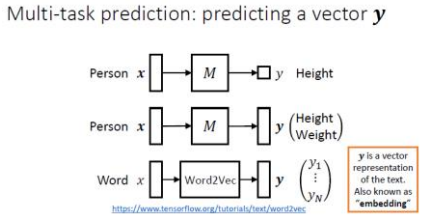
$$MAPE = \frac{100\%}{n}\sum_{i=1}^{n} \left|\frac{y_i - \hat{y}_i}{y_i}\right|$$

→MAPE measures prediction accuracy of a forecasting method

→ MSE and RMSE penalizes larger differences more than MAE.

→ RMSE allows the error to be in the same dimension as the labels as well (I think)

Vector Regression



→ Nothing much, just need to know that we can transform words into some form of text vector known as an embedding

Vector Distances and Similarity

1. Squared Distance

$d = (\text{y-hat} - y)^2$

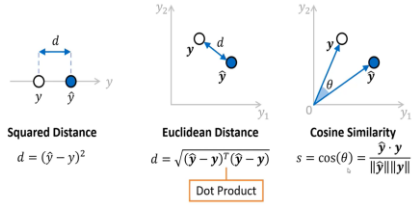
2. Euclidean Distance
3. Cosine Similarity

→ Often used for text embeddings, since their vectors are unit length

→ Smaller the angle, cos(**θ**) closer to 1

→ Measures how similar these 2 points y and y-hat are

→ In cosine similarity formula, we are taking dot product divided by multiplication of magnitude

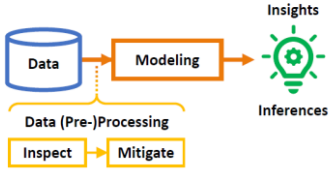


Summary

Different evaluation metrics are suitable for different prediction tasks (Classification vs regression for example)

<u>Data Processing</u>

→ Step before modelling in ML pipeline



Linear Separability

→ Use feature engineering, making non-linearly separable datasets separable to perform classification (similar to kernel trick in svm)

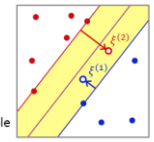
→ Issue with models w.r.t linear separability:

1. Many models assume data features are linearly separable.
2. Irrelevant features are uninformative to train model to discriminate between prediction label. If not all features are linearly separable, we cannot learn a good linear model and need to use more complex models.
3. Happens most of the time, especially for unstructured (non-tabular) data, e.g. images, time, text
4. How to check for it? Visualize data points using scatterplot.

→ 2D: Scatterplot of x1 by x2 graph

→ >2D: Scatterplot Matrix. Scatterplot for each pairwise matrix

→ If too many dimensions? Use computational metrics such as linear SVM to determine how linearly separable our model is



- Calculate total violation based on the points which has crossed the soft margins, total violation indicates how linearly separable the data is in terms of its features. Higher violation

→ less linearly separable

→ We can also reduce dimensions (LDA, PCA), then check for separability

→ Other possible ways: Linear programming, convex hulls (???)

5. How to mitigate it?

→ Find useful features (Feature extraction, hunt for other variables to create linearly separable data)

→ Transformation of features

- Feature Engineering, changing of basis vectors (PCA,LDA), kernel trick (for kernel SVM), feature learning (I think using the output hidden layer from NN?)

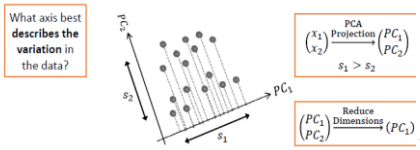
Principal Component Analysis (PCA)

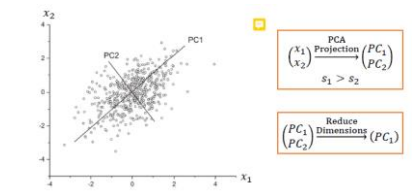
→ Which axis best describes the variation in the data? Axes are called principal components

→ Finds axis with data that has maximum variance (most linearly separable)

→ **Achieves dimensionality reduction**

→ normalize features before applying so that variances are comparable



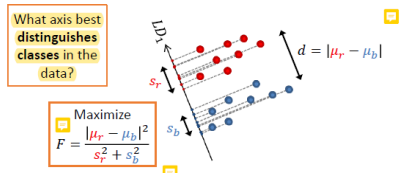


→ New axes are orthogonal (perpendicular), which means that the created features are completely unrelated. (The features were created as projections on orthogonal basis)

→ In this case, PC1 is 'more informative' since variations in PC1 can give us more information about the change of the output across the dataset

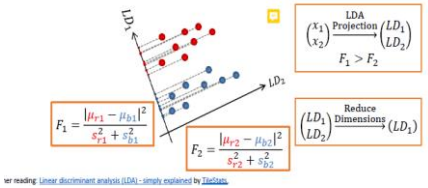
Linear Discriminant Analysis (LDA)

→ Which axis best distinguishes classes in the data?



→ Maximises F. F is the difference is squared of the average distance between 2 classes, divided by spread or scatter of each class

→ We want a bigger difference along with smallest spread to get biggest F



→ Comparing the 2 axes, F1 > F2 since the points along LD1 give a better F score than points along LD2 → We reduce the dimension to include LD1

PCA vs LDA

→ PCA: Dimensionality reduction for supervised **regression** and **unsupervised learning**

→ LDA: Dimensionality reduction for supervised **classification**

→ Steps: We assume the axes are orthogonal (independent)

1. Identify basis vectors (the axes)
2. Rank the basis vectors by importance
3. Truncate selection of basis vectors (keeping the most important features, achieves dimensionality reduction)

→ If data is not linearly separable, PCA and LDA makes it easier/faster to find optimal decision boundary by reducing non-linearly separable dimensions.

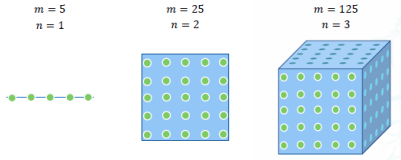
→ **PCA does Feature Extraction,** makes features less interpretable because its just finding the axes that gives the highest variance. PCA does not use labels in feature extraction

→ LDA does feature extraction, maximises component axes for class separation. Uses labels for feature extraction

Curse of dimensionality

→ Sparsity with high dimensions

Sparsity problem: Maintaining density of samples depends on exponential growth of the data



→ In high dimensional space, most points become nearly the same distance away, learners that depend on distance break down in high dimensions (KNN, SVM?)

→ Main problem? Too many features; much more features than instances

→ Why isit a problem?

1. Data becomes too sparse to inform about true decision boundary (for classification) => Too easy to fit a model on sparse training data => Overfitting
2. Distances are too similar (bad for kNN, k-means clustering)
3. When would it happen?

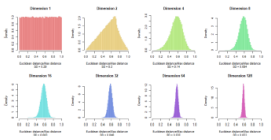
- Extract more features than data instances
- Unstructured data (features such as image pixels/sensor readings)

4. How to check for it?

- Visualize histogram of distances, check for variance σ^2

→ We usually aim for a maximum of 20% the number of features to data points (1200 features on 6000 datapoints)

- Visualize histogram of **distances** (check for **variance** σ^2)



→ Above is graph of density(y) to euclidean distance(x), as we progress to larger dimensions, most of the distances are concentrated onto a very small range of values i.e. the points are almost all equidistant from each other

5. How to mitigate?

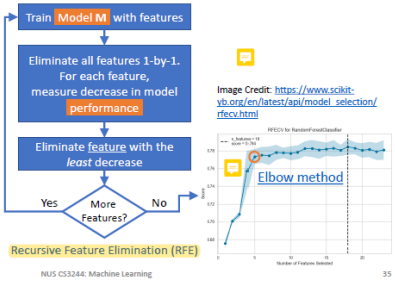
a. Feature Selection

→ Wrapper methods (RFE, recursive feature elimination)

a. Train model with features, eliminate features 1 by 1, measure decrease in model performance, eliminate feature with the least decrease.

→ Very computationally expensive

→ Can observe results using the elbow method (graph of number score/performance (y) and number of features selected (x))



→ Filter methods

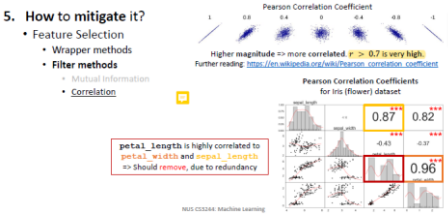
- Mutual Information = Information Gain (Remember DTs), greedy approach

→ Different from DTs in the sense that for DTs, we calculate IG on the subset of data as we traverse down the tree, but for feature selection, we always do it on the entire dataset to see which has highest IG (we take this one)

- Correlation

→ Pearson correlation coefficient (Measures the correlation between 2 features). If 2 features are very highly correlated, then we might as well just pick one, having the other is redundant

→ r>0.7 is very high

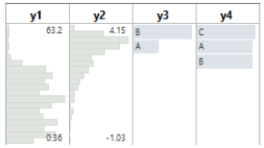


b. Dimensionality Reduction
→ PCA, LDA (Linear Matrix Factorization), Deep Auto-Encoders, Non-linear manifold learning (not tested)

Benefits of Feature Selection
- Avoid curse of dimensionality
- Faster model training (optimize fewer parameters on fewer features)
- Fewer features to read → easier to interpret

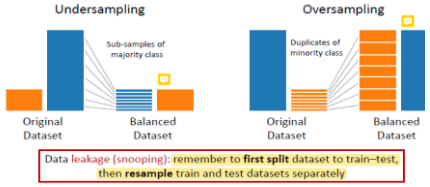
Imbalanced Data
→ For **over-represented features,** we say that the data is **skewed.** (More males than females)
→ For **over-represented labels,** we say that the data is **imbalanced.**
→ As long as some columns are visibly more than others we can say that it is imbalanced / skewed

1. What is the issue?
→ Values not evenly distributed in feature; Data is skewed
2. Why is it a problem?
→ Evaluation metrics become misleading to interpret (think accuracy, precision, recall)
→ Models overfit to majority class
3. When would it happen?
→ When events unevenly occur (rare cancer)
→ Data collection is uneven (only positive survey respondents)
4. How to check for it?
- Visualize histogram (continuous) or bar chart (discrete) of feature values



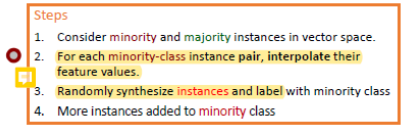
5. How to mitigate it?
- Collect more data instances (Needs money and time)
- Resample instances (Undersampling, oversampling, SMOTE)
- However, we need to consider if this is representative of the real world. If the real world is imbalanced, we should train our model to handle imbalance well

Data Resampling
Undersampling vs Oversampling



- Undersampling reduces the number of samples from majority class to achieve balanced dataset.
→ Consider only using this if we have enough data!! (tutorial 6)
- Oversampling duplicates data in minority sample to achieve balanced dataset
→ Both ways very naïve, one wastes a lot of data, the other is 'cheating' since we are duplicating data
→ Remember to also take note of data leaking (snooping); split dataset to train-test, then resample train and test datasets separately

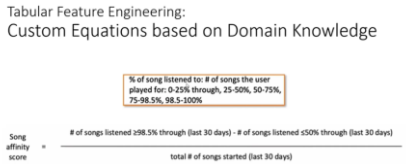
SMOTE (Synthetic Minority Oversampling Technique)



→ Assumes that interpolating the data is safe and 'correct'(?)
→ Data should be linearly separable. If a red point is somewhere in between green points, then we are synthesising red values between green, making data even less linearly separable. It might still be better than not having data points at all, but of course this isn't a great strat.
→ Should be avoided if features are strictly categorical (cannot be interpolated)

<u>Feature Extraction & Engineering</u>
→ Process of transforming raw data to improve accuracy of models
→ Captures domain knowledge, express non-linear relationships using linear models, encode non-numeric features to be used as inputs to models

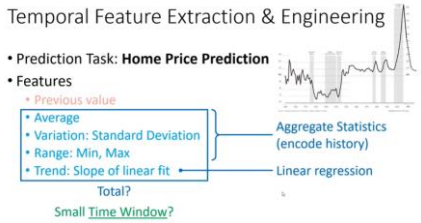
<u>Feature Eng for Tabular Features</u>
- Custom Equations based on Domain knowledge
- Derive new equations based on the knowledge, that are compatible to be fed into models (numbers)



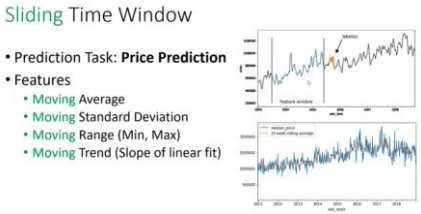
- Counting, aggregation, difference, min, max

→ Using transaction dates that was pulled from the database, generate new features

Temporal (Time) Features
- Encode history, perform aggregation on historical values

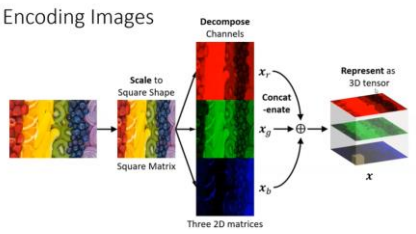


- Use sliding time window, create snapshots and perform aggregations within a sliding time window
- Convert temporal data into tabular data

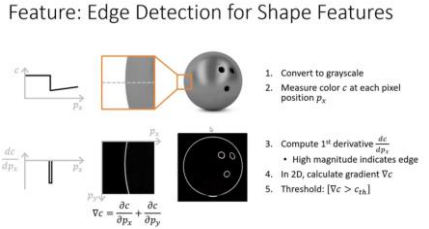


→ Can be used in sound classification as well (volume: amplitude, energy/frequency: Periodicity, spectrogram)
→ E.g. Dog barking vs drilling, there could be distinct frequencies or volumes that can be separated due to different characteristics of the sound
• **Volume: Amplitude, Energy**
• **Frequency: Periodicity, Spectrogram** → Wave analysis

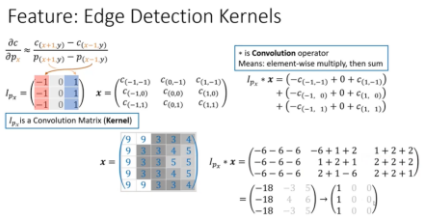
Image Features
- Encode images into square matrix (have much nicer characteristics, easier to perform learning)
- Decompose into channels (rgb)
- Concatenate them into a new dimension (like creating one more dimension in an array)



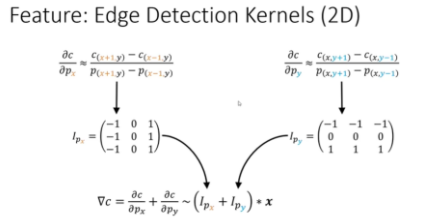
- Use color - create color histogram for the different channels and create a vector
- Shape – do edge detection (canny edge, PCA)
- Texture – Edge detection

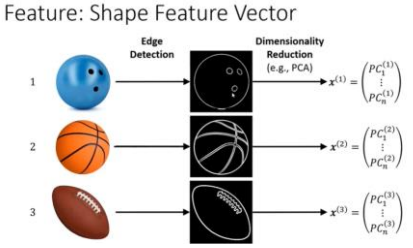


→ Grayscale, calculate gradient, using the gradient, we can identify the edge, threshold our calculations to get a clean outline of the image
→ We can also use edge detection kernels to detect edges. Perform convolution using convolution matrix on image



→ Using matrix, operations can be done in parallel, so faster on gpu
→ Will need to do convolution in x and y direction for 2 dimensional edge detection





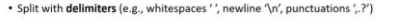
→ If we want to have something even simpler to process in our model, we can do pca on our edge detected images, getting a vectorized representation that describes the shape of the objects

Other advanced methods for images
→ SIFT (Find keypoints in image to match in other images), can create a SIFT feature vector that can be used to classify in kNN and logistic regression

Text Features

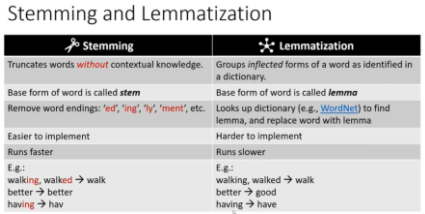
→ Stemming – take words of similar meaning and represent them the same way
→ Filter out stop words like and, this, has etc

Tokenization
→ Split substring with delimiters



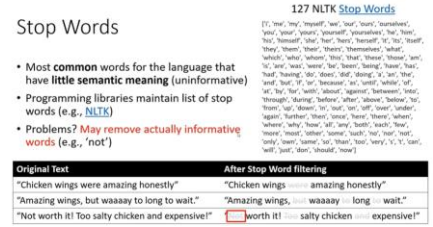
Original Text	Tokenized into Array of Words
"Chicken wings were amazing honestly"	['chicken', 'wings', 'were', 'amazing', 'honestly']
"Amazing wings, but waaaay to long to wait."	['amazing', 'wings', 'but', 'waaaay', 'to', 'long', 'to', 'wait']
"Not worth it! Too salty chicken and expensive!"	['not', 'worth', 'it', 'too', 'salty', 'chicken', 'and', 'expensive']

Stemming/Lemmatization
- Use dictionary to look up similarity



Stop Words

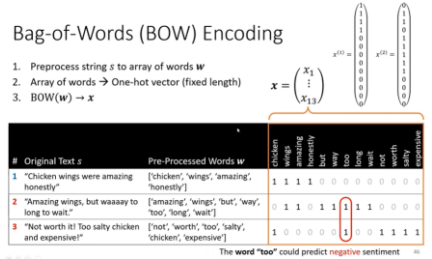
- Remove words with little semantic meaning



→ Issue: Remove words like 'not' that can flip meaning

Bag-of-words encoding

Take unstructured non-numeric data and encode into a vector



→ Split the string, take array of words and get the one-hot vector (based on the whether a certain word has appeared or not), perform BOW function to obtain vector.

→ Issue: Curse of dimensionality

Other alternatives:

N-grams

- N-grams
  - Adjacent words change meaning
  - e.g., "I am happy, not sad" vs. "I am sad, not happy"
    - But: BOW("I am happy, not sad") = BOW("I am sad, not happy")
    - With bigrams: BOW("I am happy, not_sad") ≠ BOW("I am sad, not_happy")

→ BOW will not get the difference between the adjacent words having different placings

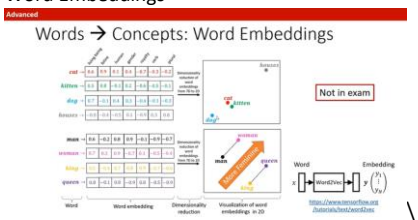
Parts of Speech

- Parts of Speech (POS) and Grammar
  - Structure of the sentence matters

→ Understand structure of sentence better (grammar, vocab etc)

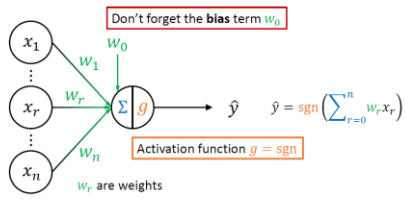
→ These help us understand why BOW is not so good

Word Embeddings



Perceptron & Neural Networks

- Linear Classifier



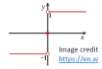
→ x1…xn – input features

→ w1…Wn – weights

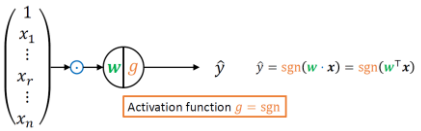
→ w0 – bias term

→ Calculate a weighted sum, pass it through the sigmoid function to get an output prediction

$$\hat{y} = \text{sgn}\left(\sum_{r=0}^{n} w_r x_r\right), \text{sgn}(z) = \begin{cases} +1 & z > 0 \\ -1 & z \leq 0 \end{cases}$$



Vector notation:

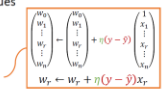


$$\hat{y} = \text{sgn}(\boldsymbol{w} \cdot \boldsymbol{x}) = \text{sgn}(\boldsymbol{w}^\top \boldsymbol{x})$$

Perceptron Learning Algorithm (PLA)

Perceptron Learning Algorithm

1. Initialize weights $\boldsymbol{w}$
   - Could be all zero, or random small values
2. For each instance i with features $\boldsymbol{x}^{(i)}$
   - Classify $\hat{y}^{(i)} = \text{sgn}(\boldsymbol{w}^\top \boldsymbol{x}^{(i)})$
3. Select one misclassified instance
   - Update weights: $\boldsymbol{w} \leftarrow \boldsymbol{w} + \eta(y - \hat{y})\boldsymbol{x}$
4. Iterate steps 2 to 3 until
   - Convergence (classification error < threshold), or
   - Maximum number of iterations

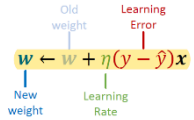


→ Weights are initialized randomly, or by all zeros

→ We need to update the weights if they are misclassified

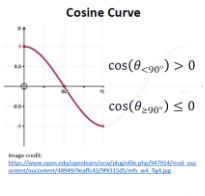
Perceptron Weight Update

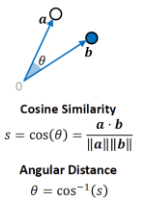
→ We take old weight as base, then based on our defined learning rate, update to the new weight



→We usually keep learning rate small so that we do not jump all over the place (see sgd slides)

Cosine Similarity (Vector Distances)





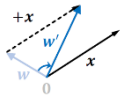
→ If angular distance < 90deg, then cosine value should be between 0 and 1. If > 90 deg then cosine value should be between -1 and 0

→ This brings us to the weight update mechanism of a perceptron

Consider the actual label y = +1, and y-hat = sgn(w.x) = -1

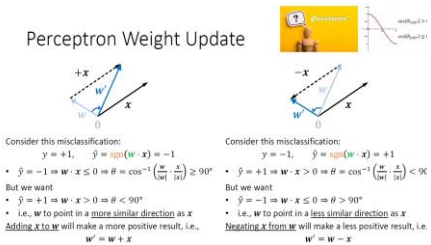
=> This means that the angle between the weight and x is more than 90 degrees in the prediction, but we want y-hat to be +1, which makes the angle between the weight and x less than 90 degrees, i.e. weight should point in a more similar direction as x.

➔ We do so by adding x to w, making w a more positive result

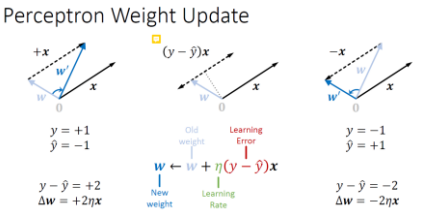


Adding $\boldsymbol{x}$ to $\boldsymbol{w}$ will make a more positive result, i.e.,

$$\boldsymbol{w}' = \boldsymbol{w} + \boldsymbol{x}$$



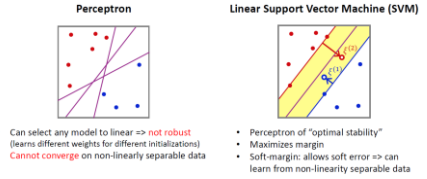
General Case:



→ by using y – y-hat, if we want x to go in positive direction, then learning error becomes positive (left e.g.), else learning error is negative (right e.g.)

→ Look at PLA on the left, **to calculate the loss, we look at each feature (x) and weight independently, then calculate the new weight based on that row only**

Perceptron vs Linear SVM



→ Perceptron, once satisfied will just return that as a correct line, unlike svm, will give optimal line that is the most stable (it maximises margin)

→ SVM allows soft error and can learn on non-linearly separably data

Extending the Perceptron

- Other **activation functions**
  - Differentiable ones!
- **Multiple** perceptrons
  - Multi-Layer Perceptron (MLP)

} Feed-forward Neural Network

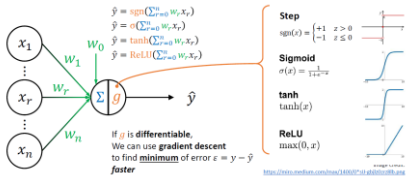
- **Non-linear** classifiers

Activation Functions

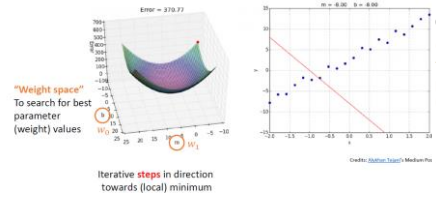
→ Most importantly, the activation function g must be a differentiable one, so that we can perform **gradient descent** to find minimum of error faster

→ Step, sigmoid, tanh, relu

→ Step sucks because the gradient is always 0 (except when x = 0, so its very uninformative)

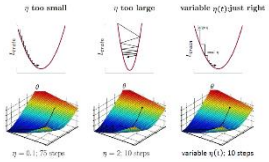


Gradient Descent (Recap)



→ Graph is in the weight space (x-axis) – its taking the different values of all weights and plotting it as a graph so calculate gradient for descent. y-axis is the error

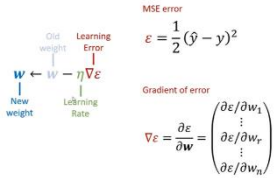
Learning Rate cannot be too large else weights will be hopping around



These graphs are also in the "weight space".

Gradient Descent Weight Update

→ Mean squared error, old weight – learning rate * learning error because we want new weight to move in direction opposite of the gradient, calculate gradient of error by performing differentiation on every single weight and its errors associated



→ Gradient descent vs perceptron weight update, gradient descent's learning error is the gradient of error, instead of just the difference between the prediction and the label

Chain Rule

→ Finding relationship between variables in composite function (in my own words)

Consider composite function

$$h(x) = h\Big(g\big(f(x)\big)\Big)$$

$$h = h(g), g = g(f), f = f(x)$$

$$h'(x) = \frac{dh}{dx} = \frac{dh}{dg}\frac{dg}{df}\frac{df}{dx}$$

→ We are iteratively trying to find Wr with minimum error e

Optimization Goal:

Iteratively find w_r with minimum error ε

$$\hat{y} = g(f(w_r)), \; f(w_r) = \sum_{r=0}^{n} w_r x_r$$

$$\varepsilon = \frac{1}{2}(\hat{y} - y)^2 \quad \text{MSE error}$$

"Weight space"

→ g is activation, f is weighted sum, error calculated using mean squared error

"Weight space"

To search for best parameter (weight) values

$$\frac{d\varepsilon}{dw_r} = \frac{d\varepsilon}{d\hat{y}}\frac{d\hat{y}}{dw_r} = (\hat{y} - y)\left(\frac{dg}{df}\frac{df}{dw_r}\right)$$

→ de/dy is only affected by the error function, i.e. MSE, MAE, cross entropy loss

→ dy/dwr only affected by number of weights (or neurons)

→ (y-hat – y) is calculated by differentiating the error equation we see above.

→ dg/df * df/dwr is calculated by looking at the y-hat equation, we see that y-hat is also dependent on the activation function and weighted sum

Weight Update

$$w_r \leftarrow w_r - \eta \frac{d\varepsilon}{dw_r} = w_r - \eta(\hat{y} - y)g'(f)x_r$$

$$\frac{dg}{df} = g'(f) \qquad \frac{df}{dw_r} = \frac{d(\sum_{r=0}^{n} w_r x_r)}{dw_r} = \frac{d(w_r x_r)}{dw_r} + \sum_{p \neq r}^{n}\left(\frac{d(w_p x_p)}{dw_r}\right) = x_r$$

→ dg/df is basically differentiating the activation function

→ df/dwr is the partial differentiation of each weight. The other weights that are not the wr in question are treated as constants when we do

the partial differentiation (So the term on the right disappears)

→ We realise that differentiating weighted sum by weight is basically the value of the feature itself (that's the gradient)

Derivative of sigmoid function

What is the derivative of sigmoid function σ?

- $\sigma(x) = \frac{1}{1+e^{-x}}$
- Rewrite as compound function
  - $\sigma(x) = \frac{1}{1+z}$, $z(x) = e^{-x}$
- Using chain rule
  - $\sigma'(x) = \frac{dg}{dx} = \frac{dg}{df}\frac{df}{dx} = \frac{-1}{(1+z)^2}(-e^{-x}) = \frac{e^{-x}}{(1+e^{-x})^2} = \sigma(x)\frac{e^{-x}}{1+e^{-x}}$
- Notice: $1-\sigma(x) = 1 - \frac{1}{1+e^{-x}} = \frac{1+e^{-x}-1}{1+e^{-x}} = \frac{e^{-x}}{1+e^{-x}}$
- Substituting back:
  - $\sigma'(x) = \sigma(x)(1-\sigma(x))$

→ This is relating to dg/df

$$\frac{dg}{df} = \sigma'(f) = \sigma(f)(1-\sigma(f))$$

→ sigmoid(f) is basically y-hat (refer to the equation above)

Weight Update

$$\frac{dg}{df} = \sigma'(f) = \hat{y}(1-\hat{y}) \qquad \frac{df}{dw_r} = x_r$$

$$w_r \leftarrow w_r - \eta\frac{d\varepsilon}{dw_r} = w_r - \eta(\hat{y}-y)\hat{y}(1-\hat{y})x_r$$

→ This is just for 1 feature

Iteratively find $\boldsymbol{w}$ with minimum error ε

$$\hat{y} = g(f(\boldsymbol{w})), \qquad f(\boldsymbol{w}) = \boldsymbol{w}\cdot\boldsymbol{x}$$

$$\varepsilon = \frac{1}{2}(\hat{y}-y)^2 \quad \text{MSE error}$$

Gradient $\nabla_{\boldsymbol{w}}\varepsilon = \frac{d\varepsilon}{d\boldsymbol{w}} = \frac{d\varepsilon}{d\hat{y}}\frac{d\hat{y}}{d\boldsymbol{w}} = (\hat{y}-y)\left(\frac{dg}{df}\frac{df}{d\boldsymbol{w}}\right)$

Weight Update

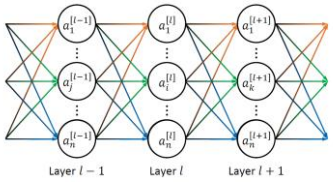
$$\frac{dg}{df} = \sigma'(f) = \hat{y}(1-\hat{y}) \qquad \nabla_{\boldsymbol{w}}f = \frac{df}{d\boldsymbol{w}} = \frac{d(\boldsymbol{w}\cdot\boldsymbol{x})}{d\boldsymbol{w}} = \begin{pmatrix} \vdots \\ d(w_r x_r)/dw_r \\ \vdots \end{pmatrix} = \begin{pmatrix} \vdots \\ x_r \\ \vdots \end{pmatrix} = \boldsymbol{x}$$

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta\frac{d\varepsilon}{d\boldsymbol{w}} = \boldsymbol{w} - \eta(\hat{y}-y)\hat{y}(1-\hat{y})\boldsymbol{x}$$

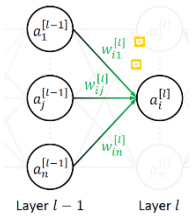
→ Nothing changes when we expand this to the entire layer, its still the same calculations done for each weight

Neural Networks

Multi-Layer Perceptrons



→ Note the notations of layers and neuron



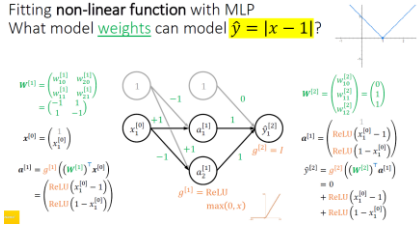
Superscript – **where am I going to** (not where im coming from)

Subscript - i1 - left is destination right is source going to i coming from 1

Example of fitting non-linear function with multi-layered perceptron

Fitting **non-linear function** with MLP

What model weights can model $\hat{y} = |x-1|$?



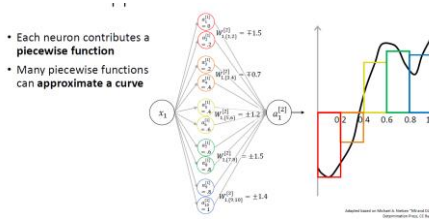
→ **Note: REMEMBER NEGATIVE WEIGHTED SUM SHOULD OUTPUT AS 0**

→ Try to use differentiable functions for activation, if we have non-differentiable, differentiable * non-differentiable is non-differentiable, so it gets worse and worse

Universal Approximation Theorem

→ Each neuron contributes a piecewise function

→ Each piecewise function can approximate a curve



→ Each bar is an output from one neuron

Backpropagation

Backpropagation **efficiently** computes the **gradient** by

- Avoiding duplicate calculations
- Not computing unnecessary intermediate values,
- Computing the **gradient** of ***each*** **layer**

→ We calculate the gradient from layer [l + 1] to front [l]A

Specifically, the gradient of the weighted input of each layer is calculated from back $[l+1]$ to front $[l]$:

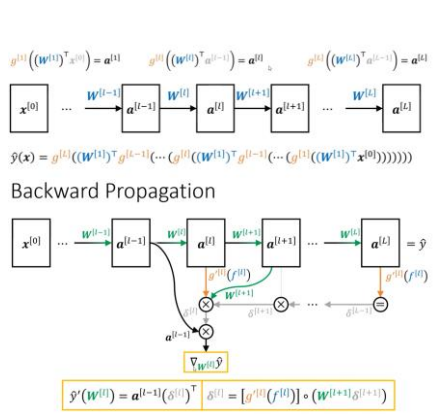
$$\hat{y}'(W^{[l]}) = \boldsymbol{a}^{[l-1]}\delta^{[l]} \qquad \delta^{[l]} = g'^{[l]}(f^{[l]})W^{[l+1]}\delta^{[l+1]}$$

Layer l Activation Function — Layer l Weights

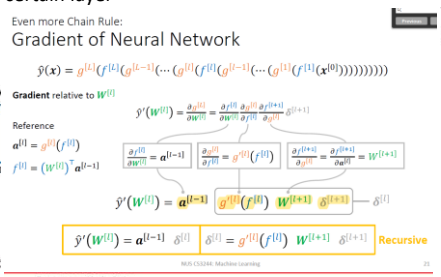
$$\boldsymbol{a}^{[l]} = g^{[l]}\left((W^{[l]})^{\top}a^{[l-1]}\right)$$

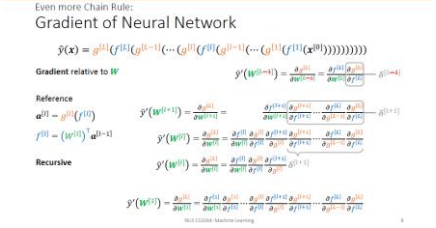
Layer l Activations — Layer $l-1$ Activations — Layer l in Neural Network

→ We need to understand fwd prop before back prop. a[1] is calculated using the initial weights x[0], subsequent a[l] uses a[l-1]



→ Gradient output with respect to weights at current layer l is given by the product of the activations of the previous layer and the g,f and W of future layers. That's how we calculate the how y changes with respect to weights at a certain layer





Math Notations

Notation

n = Number of features in x
m = Number of instances in dataset

- **Scalar**: not bolded, lower case x
- **Vector**: bolded, lower case $\boldsymbol{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$
- **Matrix**: bolded, upper case $\boldsymbol{X} = \begin{pmatrix} x_{11} & \cdots & x_{m1} \\ \vdots & \ddots & \vdots \\ x_{1n} & \cdots & x_{mn} \end{pmatrix}$

Deep Neural Network

→ >=3 hidden layers

- Target functions of real-world tasks are complex, require many parameters and layers to model

→ Requires a lot of training data, as there are many parameters. More data → better perf

Traditional ML: Requires us to do feature extraction/engineering…

Deep Learning: Automatic feature learning!!!

CNN (Convolutional NNs)

- Use cases: Image Classification, Object Detection, Image Segmentation (cancer cell detection)

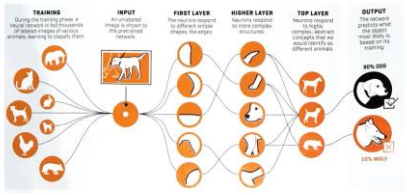
→ Images are typicaly 2D matrices (eg 256 x 256)

→ We can simply flatten a 3x3 image to a 9 x 1 image to perform Fully Connected NN (Multi-layer perceptron) on it

→ But we wont be capturing spatial relations in an image → Exploit spatial relations with convolutions!

→ At early layers, learn 'simple' features like edge detection, next layers learn more 'higher' concept entities like shape of a nose, ears etc. This is possible because our NN is compositing the lower level features into higher level ones

Feature Detectors: Intuition of Neuron Kernels in Layers

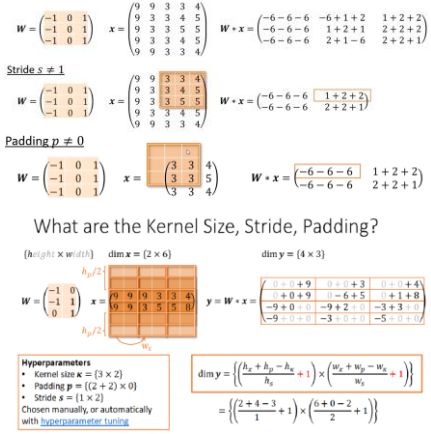


→ Convolutions are like filters to allow us to take an image and represent them in different ways

Convolutions: Kernel Size, Stride, Padding

Stride: How much my convolution 'shifts' per step

Padding: Allow our convolution to move 'beyond' the image



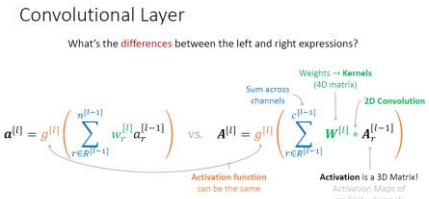
→ If dim y returns a decimal, we round down. Since we have already taken padding into account and any loose rows wont be convulated over I think?

→ Take note how to multiply a convolution over an matrix, we do Hadamard operator so we match the position side by side

→ K = { 3 x 2 } means 3 vertically, 2 horizontally. Applies for p and s as well.

→ Notice the dimensions of the padding is calculating the amount of 'overhang'

CNN vs Traditional NN



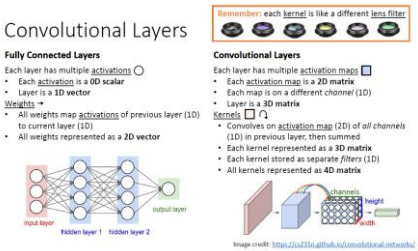
→ activation function g is the same

→ weights is a vector (1D matrix) in NN, a 3d matrix (kernels) in CNN

→ Activation A is a vector in NN, a 3D matrix in CNN

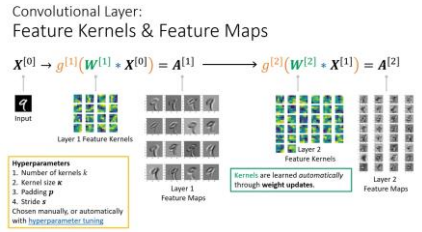
→ Jargon: Convolution produces a **feature map**, weights are called **kernels**

Convolutional Layers Vs Fully Connected Layers



→ Into one kernel, we have many neurons coming into it (activation maps, each 2D). These activation maps will be convolved against kernels. (These kernels are 3D, because we split the kernels to have different channels). **Why can we convolve a 3D kernel onto a 2D feature map?** By convolving each 2D feature map with each layer of the kernel (3D sliced into 2D). We get a 3D matrix, which we sum together to get a 2D feature map again.

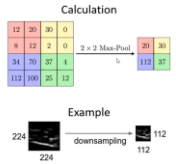
→ All kernels represented as 4D



→ Take note of hyperparameters in doing CNN
→ Referring to diagram, there are 16 feature kernels in layer 1, means 16 neurons, so for each neuron, we take input multiply weight for a feature map in each neuron.
In layer 2, there are 32 kernels so 32 neurons, so for each neuron, we take the 16 kernels from previous layer (3d stacking of feature maps), multiply with my kernel. We will be getting a 16 * 256 * 256 sized map per neuron, which we **sum** together to get one feature map and that's why in layer 2, we have another 32 feature maps again (1 feature map per neuron)

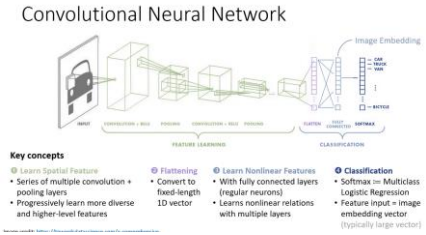
Pooling Layer

- **Downsamples** Feature Maps
- Helps to train later kernels to detect **higher-level** features
- Reduces **dimensionality**
- Aggregation methods
  - Max-Pool (most used)
  - Average-Pool
  - Sum-Pool



→ Max pool takes the maximum pixel value of the 4 in that pool

CNN Pipeline



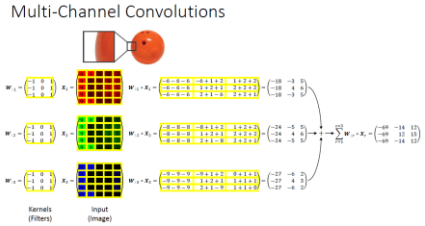
→ Convolution + Pooling is basically learning the spatial features. We then flatten so that we can perform classification with a neural network. Flattening is ok here because each output map is already more 'semantic', and has captured a pretty good meaning from the images
→ Use fully connected layers to learn because it helps us to learn nonlinear relations/functions. Usually its just 1 fully connected layer.
→ Then we use softmax to classify (which is basically a multiclass logistic regression)

→ Take note the layer after using fully connected layer is called an image embedding

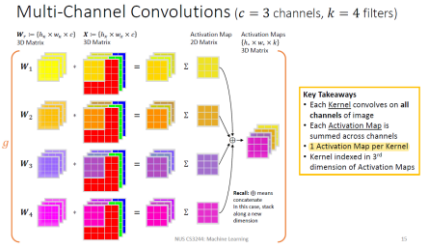
Advantages of Batch Image Training

- Used in many deep learning libraries
- Computationally efficient, and more stable in gradient descent convergence
- If input is H x W x C (number of kernels) on B number of images, output is B x H x W x C

Multi-Channel Convolutions



→ Convolve through each channel, we get 3 separate matrices which we **sum** together to get one single matrix

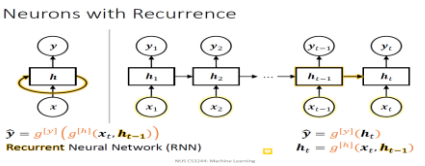


→ After the multichannel convolution, we stack the feature maps together based on the number of kernels we have (W1 – W4) and pass the k-layered matrix through activation
→ So my current layer's weights are called kernels, but the weights that was used for my previous layer are called **channels**

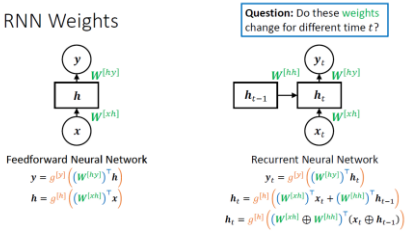
RNN (Recurrent Neural Networks)

Use cases:

RNN Architecture

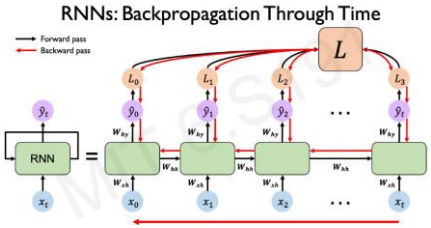


→ **We want some way to connect the hidden layers at each time step to each other**, because an output like Yt in a time series should have been influenced by ht-1.
→ Why don't we take the inputs x1,x2… as the input for the future layers instead? For conciseness. Since the hidden layers are basically a 'representation' of the input Xs, we can simply use hidden layer h as a more concised version that is passed onto the next layer
→ y-hat is a function of ht, and ht is a function of xt and ht-1



→ Every input will have its associated weights
→ Notice the more concised notation of ht (the second one). (Thinking about it im not really sure why it works tho, it feels like it shouldn't be equivalent)
→ Doesn't change with time per se, it just depends on the previous hidden layer and the current input

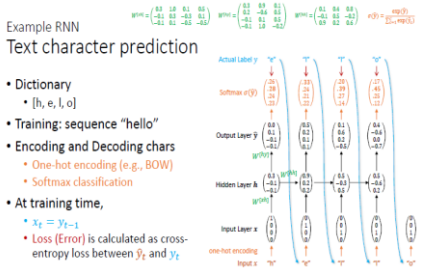
Backpropagation in RNN: BPTT



→ The paths that were taken forwards can also be used to perform backprop. (Seems like theres shortcuts since every step will give you one output as well)

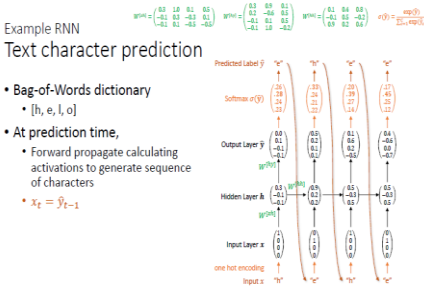
Text Representation

We need numbers/vectors in our RNN inputs, so how do we translate text into numbers?
- Encoding and Decoding chars using one-hot encoding (e.g. BOW) / softmax classification



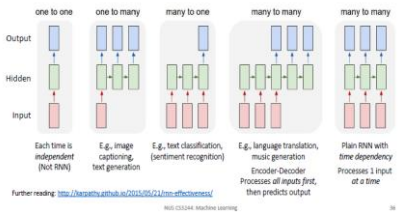
→ Using one-hot encoding, in this example, we can take it as if our universe has only 4 characters. With only 4 characters, how do we translate the numerical output into a letter again? → Can use softmax to get a prediction again. Output y-hat will be used as input for next layer, xt.
→ Cross entropy loss calculated between prediction y-hat(t) and y(t)

→ **At prediction time, it has no labels to refer to and cannot rely on the actual labels to predict**.



→ RNN is purely **using predicted label** to predict future labels. Cant cheat and take a peek when testing!

Other Sequence Modelling Applications



→ One to many: Give it one thing, predict multiple things, Many to one: Give it sentences, predict a word (sentiment for e.g.)
→ Other many-to-many applications: Listens to everything, after listening to everything, then output music (for e.g.)

Characteristics of RNN

→ Captures sequential interdependencies between inputs.
→ Works for sentences, time series kind of input, works well with different length (sized) inputs
→ The classic RNN tends to model recent past time steps well better than distant ones, as the parameters from the recent inputs have been 'compounded' on less

Deep Learning Training Issues

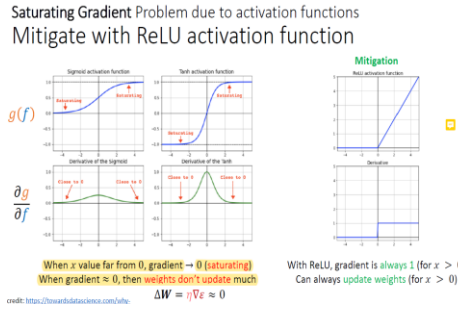
- Overfitting
- Saturating Gradient Problem, Vanishing Gradient Problem

Overfitting in Deep Learning

→ Deep learning has too many parameters! It will fit to the training set too well if there are too many parameters in place.
Mitigation: **Dropout**
→ During forward propagation and back prop, randomly drop out some neurons during batch training, so we cannot propagate through those neurons during training.
→ **Important:** All nodes are still used for prediction!
→ **Reasoning:** Some of the nodes in the weights were trained to be 'insensitive' to the training data, value in the node is more 'raw' (less processed), reduce chances of overfitting (might reduce overfitting but sounds to me like it might increase bias as well, how do we know the randomly initialized value in that node is a good value??)

Saturating Gradient Problem

→ Consider how we do backprop to update our weights: We have to different dg/df (differentiating the activation function) in order to update weights.



→ Gradients saturate (don't change much) when x is far from 0 => very hard to perform weight updates! (Even though far from 0 means model is quite confident of its answer, NN is full of parameters and one 'confident' output can greatly reduce the amount of learning of all parameters, since we rely on future values in doing back prop)

Mitigation: Use ReLU, gradient is always 1 for x > 0, and can always update weights for x > 0

What does it mean if ReLU outputs 0?

- Nothing gets backpropagated through them, and if every possible training input returns ReLU(x) = 0, it takes no role in discriminating between inputs! So its like a 'dead' neuron.

Vanishing Gradient Problem

- Recall that weight change = learning rate * gradient of error.
- When we backprop, if some gradients are very small, multiplying many small numbers equals to a very small number → earlier layers do not get updated much.

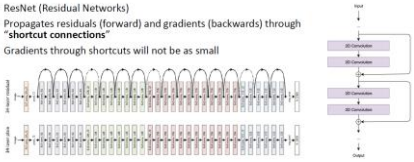
→ This is why a function like sigmoid might not work so well. As long as the gradient of something in the expression

$$\hat{y}'(W^{[1]}) = \frac{\partial g^{[L]}}{\partial W^{[1]}} = \frac{\partial f^{[1]}}{\partial W^{[1]}}\frac{\partial g^{[1]}}{\partial f^{[1]}} \cdots \frac{\partial g^{[l]}}{\partial f^{[l]}}\frac{\partial f^{[l+1]}}{\partial g^{[l]}}\frac{\partial g^{[l+1]}}{\partial f^{[l+1]}} \cdots \frac{\partial f^{[L]}}{\partial g^{[L-1]}}\frac{\partial g^{[L]}}{\partial f^{[L]}}$$

Is < 1, it will reduce the gradient significantly. dg[l]/df[l] = derivative(activation), df[l]/dg[l-1] = derivative of weighted sum = a[l-1].

If some gradients are small (< 1), multiplying many small numbers equals a very small number. E.g., $0.5^{15} \approx 0.0003$

Mitigations: ResNet, Have shortcut connections to allow forward and backward propagations through these connections, so gradients do not shrink as much!



Summary

→ With NNs, we do not have to do manual feature engineering, but we have to do architecture engineering, to figure out how to best design a neural network for your problem (num of layers, num of neurons, shortcut connections etc)

XAI (Explainable AI)

- Provide explanations to predictions in the context of our data

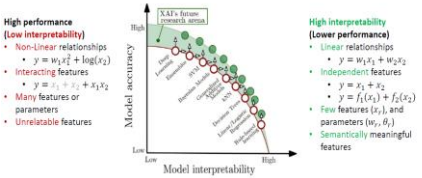
Feature Importance: explains which features are important for the prediction, in what way the features influenced the predictions

Feature Attribution

→ Based on some form of graph, have a chart that expresses which attributes contribute to the prediction greatly/minimally

Performance-Interpretability Tradeoff

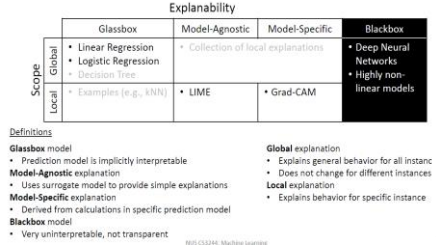
→ Either have high performance low interpretability or vice versa



High interpretability: Linear relationships, features that are independent from each other, few features/parameters

High Performance: Non-linear relationships, features that interact with one another, many features/parameters and unrelatable features

XAI Categorization



→ Take note of global vs local, global explainability explains the reason for a prediction in the same way for every input, local is different depending on the data point's values

Interpreting Linear Regression

→ Recall LR is basically weighted sum

$$\hat{y} = w_0 + w_1x_1 + w_2x_2 + \cdots + w_nx_n = \sum_{r=0}^{n} w_rx_r = \boldsymbol{w} \cdot \boldsymbol{x} = \boldsymbol{w}^T\boldsymbol{x}$$

$\hat{y} \propto w_rx_r, \forall r$ — If $w_1 = kw_2$, then w_1 is k times more important than w_2

Weighted Sum Interpretation	Gradient Interpretation
Bigger w_r means • **Larger** weight • More **importance** for to x_r • Direction? Supportive (positive) or opposing (negative) **influence**	**Bigger w_r means** • **Steeper** slope for x_r axis • Changes in x_r lead to bigger in $\hat{y}$ changes • More **importance** for x_r • Direction indicates increasing or decreasing **influence**

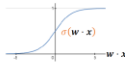
→ We can explain based on weights, more weight → higher importance, or by gradient, More weight = steeper slope so changes in that attribute contribute greater to the prediction of y.

Interpreting Logistic Regression

Not proportional, since non-linear. If $w_1 = kw_2$, then what is the relationship between w_1 and w_2?

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

$$z = \boldsymbol{w} \cdot \boldsymbol{x} = \sum_{r=0}^{n} w_rx_r$$



$\hat{y} \not\propto w_rx_r, \forall r$

Weighted Sum Interpretation	Gradient Interpretation
Bigger w_r means • **Larger** importance • Direction indicates **influence**	**Bigger w_r means?** • Steepness? Sigmoid bounded between 0 and 1 • Direction in 2D (or higher)?

$\hat{y} = \sigma(f)$ is positively monotonic, i.e., $f_1 > f_2 \Rightarrow \sigma(f_1) > \sigma(f_2)$. Bigger $f \Rightarrow$ bigger $\hat{y}$

→ Logistic Regression uses sigmoid, so we cannot use 'proportional'

→ monotonic – the relative ordering is kept in this function, if f1 > f2, after passing through sigmoid, this condition still holds

Use odds ratio to interpret weights in logistic regression

Probability

$$P(\hat{y}) = p = \hat{y} = \frac{1}{1+e^{-\boldsymbol{w}\cdot\boldsymbol{x}}} = \sigma(p)$$

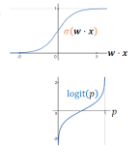
$$\frac{1}{p} = 1 + e^{-\boldsymbol{w}\cdot\boldsymbol{x}}$$

$$\frac{1-p}{p} = \frac{1}{p} - 1 = e^{-\boldsymbol{w}\cdot\boldsymbol{x}} = \frac{1}{e^{\boldsymbol{w}\cdot\boldsymbol{x}}}$$

Odds Ratio

$$\frac{p}{1-p} = e^{\boldsymbol{w}\cdot\boldsymbol{x}}$$

Log Odds Ratio $\text{logit}(p) = \log\left(\frac{p}{1-p}\right) = \boldsymbol{w}\cdot\boldsymbol{x}$



$\text{logit}(P(\hat{y})) \propto w_rx_r, \forall r$ — If $w_1 = kw_2$, then log odds ratio of w_1 is k times bigger than of w_2

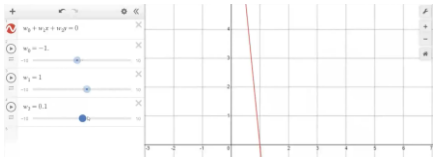
→ Recall the predicted label of log regression is like a probability of classifying an input with the output

→ Odds ratio: Ratio of something happening against it not happening (p / 1 - p)

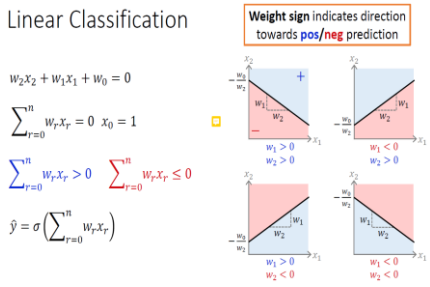
→ Weights are proportional w.r.t log odds ratio in log regression!

→ Observe logit(p) graph: If p is high, ratio shoots to infinity, If p is low, ratio shoot to negative infinity

Linear Classification



→ When w2 is very small, the line is almost perpendicular to w1(x-axis), i.e. a small change in w1 will greatly affect the output! This applies for logistic regression as well.



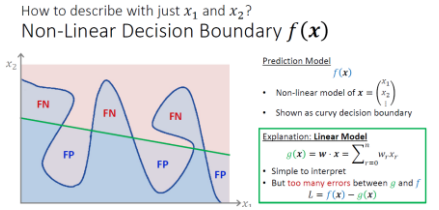
→ This diagram tells you the how the weights will determine with the input x1 and x2, what is the output of the equation.

→ How we can think about it: w1 > 0 means if u go towards positive values of axis it gets more blue. w1 < 0 means if u go towards negative values of axis it gets more blue

LIME (Local Interpretable Model-Agnostic Explanations)

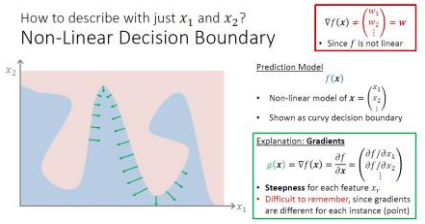
→ Explains a prediction with respect to its local boundaries

Naïve Case: Use a linear model to explain a decision boundary



→ Easy to interpret but very simply and a lot of errors (Look at all the FNs and FPs)

Take 2: Use gradient at the current point as an explanation

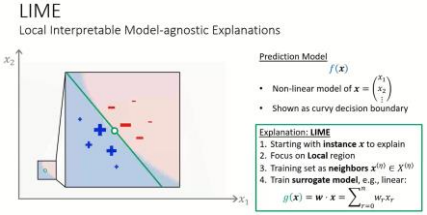


→ We will need to calculate the partial derivatives of each feature, then record the gradients in a vector. We use this vector to explain how the prediction was made.

→ However, gradients can change a lot in values even by slightly pertubing a point. Makes the explanations very hard to remember, since gradients are different for each instance.

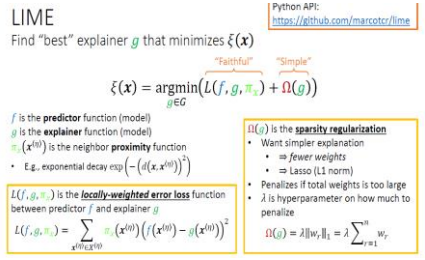
LIME

- Pick a data point, focus on its neighborhood and train a model by picking its neighbors and taking it as a training set.
- Train a **surrogate model** (e.g. linear model, log regression model)
- The model we train is a 'very good' estimation of the boundary (since its just within a small neighborhood)



→ The nearer + and – are trained with more carefully in determining the boundaries

LIME finds the best explainer g that minimizes the loss function:



→ argmin: go through all the gs, find me the g that gives me the smallest value

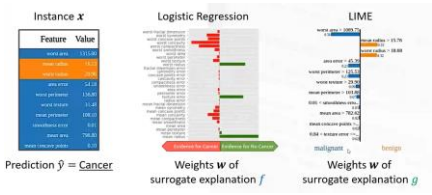
→ neighbor proximity function: exponential decay; if youre closer then you have a bigger influence, further away has less influence

L: Locally-weighted error loss $L(f,g,\pi_x)$
→ $(f(x^{<\eta>})-g(x^{<\eta>}))^2$ – find loss between label and explanation, calculate loss if they don't agree. Squared because we don't care if the error is positive or negative, and also to penalize large errors more
→ $\pi_x(x^{<\eta>})$ is to calculate **proximity**, if you are far away, then I don't really care about the error, but if youre close by, then I care more and penalize more
→ This is iterated through every neighbor sum($x\eta \in X\eta$)

Sparsity Regularization $\Omega(g)$
- We want fewer weights for a more interpretable explanation. "Just tell me the explanation in terms of top 5/10 variables"
→ Use L1 norm to achieve (remember that using L1 norm will lead to choosing less features)
- Penalizes if total weights is too large
- λ is hyperparameter on how much to penalize

$$\Omega(g) = \lambda\|w_r\|_1 = \lambda\sum_{r=1}^{n} w_r$$

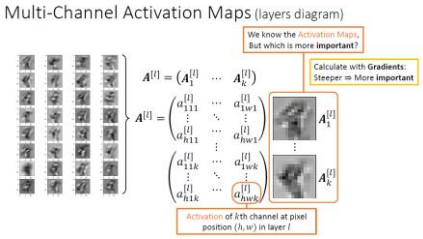
→ Lime is 'faithful' (explainer predicts same as actual prediction) and 'simple' (use of regularization)
→ Can be computationally expensive when trying to expensive multiple data points
→ "post-hoc" explanations



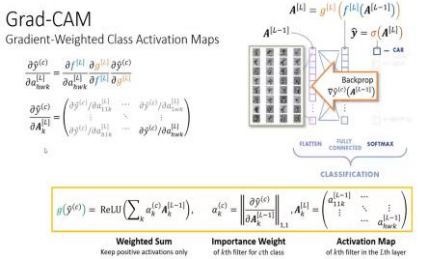
→ The LIME one is a **local** explanation, whereas the logistic regression one is a **global** explanation
→ They can be very different

Cons of LIME
- Hard to find proper neighborhood, need to find new kernel widths for every neighborhood
- Repeating sampling process can result in different explanations
- Locally perturbed data may not be realistic

Grad-CAM (Gradient-weighted Class Activation Mapping)
- Imagine using LIME to explain image predictions – it is model-agnostic, so it only knows about the inputs and what is the output (i.e. for images, it's the pixels) → Too many features!
- Our activation maps from CNN are 'features' – Grad-CAM makes use of those to produce a saliency map/heatmap
- Remember we get a lot of activation maps (each neuron), so we want to know which activation maps are more important in getting the prediction



→ Gradients – steeper = more important, we calculate the gradients in each map

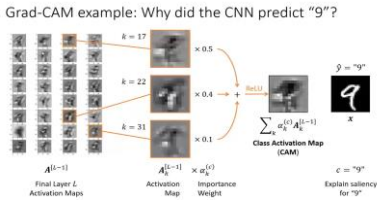


→ **Activation Map eqn:** We find the activation maps before they went through the fully connected layers (remember in CNN, we have convolutions → flattening into fully connected layers)
→ **Importance weight eqn:** We want to calculate the partial derivative of the prediction with respect to **1 activation pixel.** We do this for every single pixel, then add each individual derivative calculated (L1 norm) to get importance weight, **to know collectively** how important is the activation map. **Normalize important weights to 1 (see e.g. on right)**
→ **Weighted Sum eqn:** $A_k^{[L-1]}$ is the activation map, and $\alpha_k^{(c)}$ the importance weight associated with that activation map. Multiplying them together is like taking a 'weighted sum' which we pass through relu to tell us how much contribution that activation map had in our prediction. "For this class, tell me the weighted sum of all the heatmaps", size of matrix = 2D matrix with heigh and width of last layer of convolution

Grad-CAM Steps

1. Compute Activation Maps $A^{[L]}$ of last conv layer L
   1. via Forward Propagation
2. Choose class label c to explain about (e.g., predict "9", "car")
3. Filter prediction $\hat{y}$ to be about class c
   1. Given: $\hat{y} = \begin{pmatrix}\hat{y}^{(1)}\\ \hat{y}^{(2)}\\ \hat{y}^{(c)}\\ \hat{y}^{(n)}\end{pmatrix}$, $e^{(c)} = \begin{pmatrix}0\\0\\1\\0\end{pmatrix}$, then $\tilde{y}^{(c)} = \hat{y} \circ e^{(c)} = \begin{pmatrix}0\\0\\y^c\\0\end{pmatrix}$
   2. To generate explanation only for that class c
4. Compute importance weight $\alpha_k^{(c)}$ for each Activation Map $A_k^{[L]}$
   1. Backprop from $\tilde{y}^{(c)}$ to get gradients at last conv layer
   - Note that gradient is relative to activations, not weights
5. Compute weighted sum with ReLU to get **Class Activation Map**

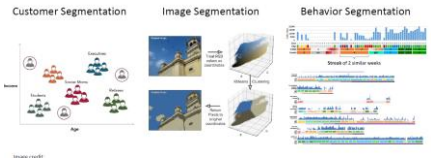
→ Take note: our gradient is activation! Not weights. Similarly when we were talking abt steepness of gradients in interpreting logistic and linear regression, the denominator were also weights. "How much a change in x affects the prediction"



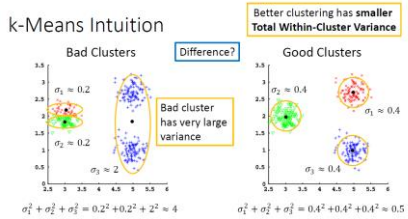
→ Sum activation maps into a ReLU
→ Instance based (local) prediction, since we pass the input through the entire model and get the activation maps for that output
Issue: Grad-CAM tells us where the model looked, but now **why** it looked over here

<u>Unsupervised Learning</u>
- Learning without any labels
- K-means clustering, auto-encoder

K-Means Clutering
Use cases: Customer segmentation, Image segmentation, behaviour segmentation



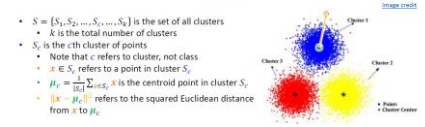
K-Means Intuition



→ Good clustering vs bad clustering: good cluster has smaller within-cluster variance, bad cluster has very large variance
→ Each cluster should have small variance
→ Sum of variance of all clusters should also be smaller

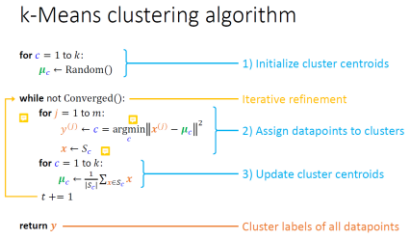
K-Means Objective
→ Minimize within-cluster sum-of-squares (WCSS), i.e. **variance**

$$L = \arg\min_{S} \sum_{c=1}^{k} \sum_{x \in S_c} \|x - \mu_c\|^2$$



→ From formula: minimize the squared sum of each point to its centroid
→ Each class = S1,S2,S3. S is set of all clusters.
→ Sc is the cth cluster of points, c = cluster
- $x \in S_c$ refers to a point in cluster S_c
- $\mu_c = \frac{1}{|S_c|}\sum_{x \in S_c} x$ is the centroid point in cluster S_c

→ We must use squared euclidean distance from a point and a centroid, because squared euclidean distance calculates variance, which is the objective we want to minimize
→ Cannot use Manhattan distance. If we want to use other distance metrics, use another clustering method called k-medoids.



Algorithm:
1. Initialize k random cluster centroids
2. while we have not converged,
 2a. assign each datapoint to the nearest cluster (by taking the centroid that returns the smallest euclidean distance)
 2b. Update each centroid by calculating the average of all points belonging to this centroid
 2c. repeat until converged

How to choose k (number of clusters)?
1. Use domain knowledge
2. Note the k with diminishing return (elbow method)
- Note the k with *diminishing* return
  - When k is too high, marginal decrease in **within-cluster sum-of-squares** (WCSS)
  - How to see?
    - "Elbow" method

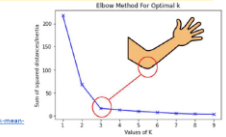


Image credit: https://www.analyticsvidhya.com/blog/2021/05/k-mean-getting-the-optimal-number-of-clusters/

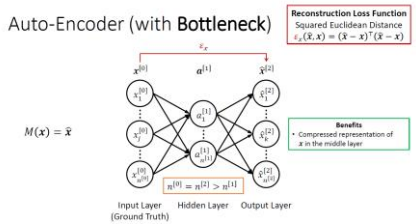
K-Means vs KNN
- What's the difference?

	k-means Clustering	k-Nearest Neighbors (kNN)
Learning Paradigm	Unsupervised	Supervised
Purpose	Group neighbors	Label based on neighbors
k is ...	Number of clusters	Number of neighbors
Distance metric	Only squared Euclidean (to match Variance)	Any distance metric (e.g., Euclidean, Manhattan, Cosine)
Measures distance between	Training set x points and cluster centroids	Test set x points and training set neighbors
Need model training?	Yes	No

Issues with K-means clustering
→ Performance of k-means sensitive to centroid initialization. Ways to improve:
- Trying multiple initializations and choosing one with minimum Lclust.
- Increase k, then manually merge clusters after k-Means.
- Use k-Means++. Seach k-means++

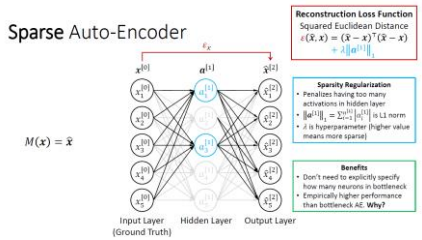
Other clustering methods (not in exam)
K-Medoids Clustering, Hierarchical Clustering, Gaussian Mixture Model, Density-Based clustering

Auto-Encoders

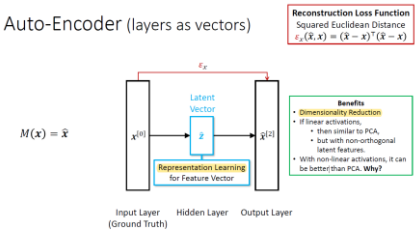


→ From input, we create hidden layers that hold less information than the inputs, then train a model that can reconstruct the input as an output from the hidden layer
→ $n^{[0]} = n^{[2]} > n^{[1]}$
→ Uses reconstruction loss.

Sparse Auto-Encoder

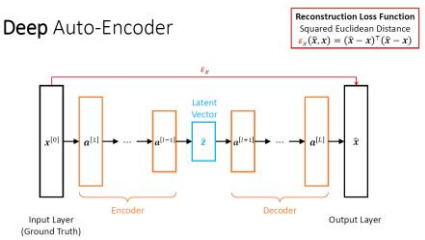


→ Not designed to be bottlenecked, but we penalize the model if too many activations are used in the hidden layers
→ Higher lambda – more sparse (more regularization)
→ Add regularization term to cost function (lasso norm)
→ Eqn in point 2 of blue box: How many terms I have in my hidden layer
→ Benefits: Higher performance than bottleneck AE! Why? Sparse AE has more flexibility, can still use more hidden neurons for training, less biased, can learn the best features, can optimize for a dataset how many neuron it needs



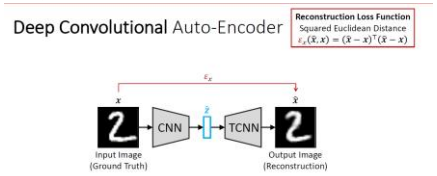
→ Hidden layer also known as **latent vector**, contains latent information of x
→ Representating learning, has learnt how to represent the data with a smaller vector
→ Benefit of auto-encoder: **Dimensionality Reduction, similar to PCA** if we use linear activations.
→ Can also be better than PCA if we use non-linear activations (like sigmoid maybe?), if the data is not linearly separable, and could've been made separable if non-linear activations were used.
→ **Caveat of Auto-Encoders:** Need more data to train since it is more complex

Deep Auto-Encoders

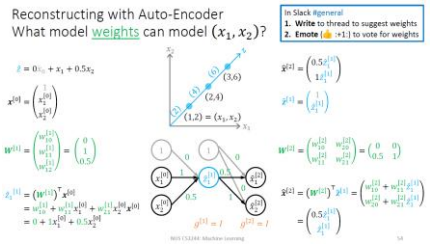


→ Have many layers in between input/latent and latent/output.
→ Layers before latent vector known as encoder, layers after latent vector known as decoder
→ More layers means we can do more non-linear transforms to get more separable data

Deep Convolutional Auto-Encoder

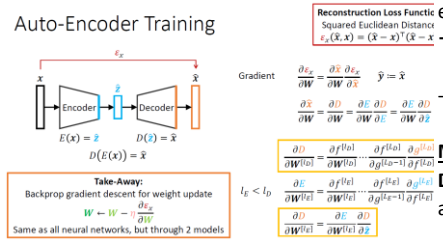


→ Just a good to know



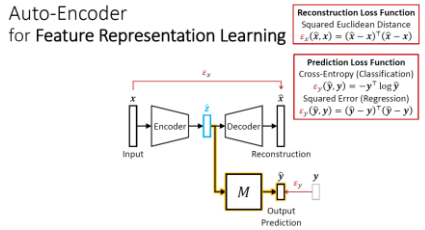
- Example of auto-encoder with associated weights

Auto-Encoder Training



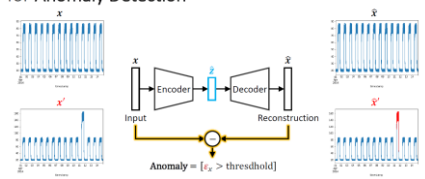
→ Before latent vector: One NN, after latent vector, another NN
→ Same gradient descent, but take note its through 2 models now, given by the equation at the bottom right

Applications of Auto Encoders



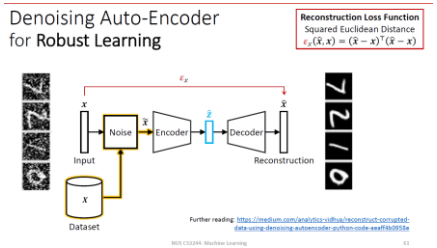
→ Use latent vector for automatic feature learning (feature representation learning), then use the latent vector to do prediction

- Anomaly Detection (Details not in Exam)



→ Error is greater than threshold, identified as anomaly

- Auto-Encoding for denoising for Robust Learning

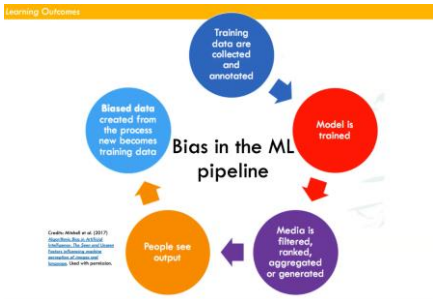


→ Add noise to input and perform auto-encoding
→ Model learns to ignore the noise

- Others: Data synthesis

ML Ethics

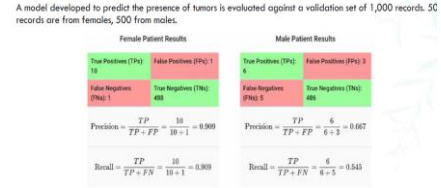
Data Bias – Bias can come into the model at anytime throughout the ML pipeline process



→ Aim to mitigate this problem
Reporting Bias – Bias in reporting data
Selection Bias – select data that seem to be the most representative; but may not be the case
Automation Bias – We might believe ML models are more impartial as compared to humans, but if we feed biased data in → it will make biased predictions
→ Results in harms of representation and allocation.
Ethics: Remove or control for sensitive fields in training, ensure that outcomes are fair with respect to sensitive fields/subpopulations, and dialogue with stakeholders: technology is not always the solution

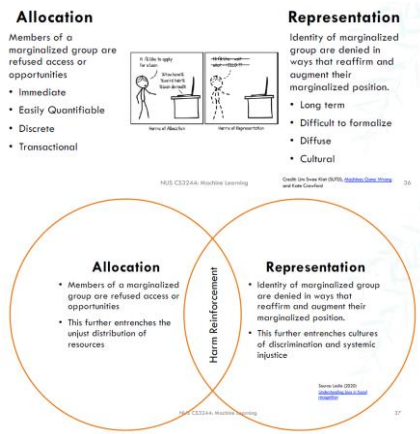
Missing and Bad Data: data collected can be inaccurate, manipulated easily, incomplete, undermined by racial bias.
Feed predictive tool contaminated data → produce polluted predictions

Politics of Cost Functions
→ Cost Function – giving a model a signal on what to optimize. We need to think deeper on its implications



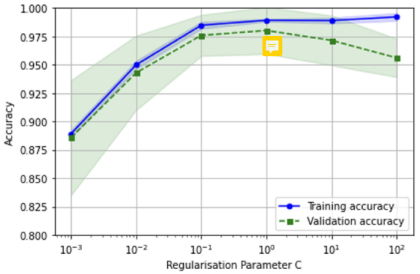
→ Example here: Precision and Recall for females much higher than males. If we look at data closely, it seems like not a very big deal since numbers are small, but this algorithm could potentially hurt males more than females, we should be more careful when we consider about fairness in the real world

Harms (Allocation, Representation)



Reinforcing Loop
- People more likely to assume algorithms are objective or error-free
- Algorithms more likely to be implemented with no appeals process in place
- Algorithms are often used at scale, are cheap and are trade secrets: no way to introspect them.

Kate Crawford
- leading scholar of social implications of artificial intelligence



(a) What does each blue and green point represent? How are they calculated?
(b) What do the blue and green shaded areas represent? How are they calculated?
(c) What should you select as your $\mathcal{C}$ value?

(a) What does each blue and green point represent? How are they calculated?
*Each blue point represents the **average** training accuracy for a value of $\mathcal{C}$. It is calculated by getting the average accuracy of all 10 training folds. Similarly, each green point represents the **average** validation accuracy for a value of $\mathcal{C}$. It is calculated by getting the average accuracy of all 10 validation folds.*

(b) What do the blue and green shaded areas represent? How are they calculated?
*Each blue point represents the **variance** of the training accuracy for a value of $\mathcal{C}$. It is calculated by getting the variance of accuracy of all 10 training folds. Similarly, each green point represents the **variance** of the validation accuracy for a value of $\mathcal{C}$. It is calculated by getting the variance of accuracy for all 10 validation folds.*

(c) What should you select as your $\mathcal{C}$ value?
The validation accuracy is the highest when $C = 1$.

Non-Linear Transformations

4. **Nonlinear Transformations.**
Consider the feature transform $\Phi(1, x_1, x_2) = (1, x_1^2, x_2^2)$. Draw and show the boundary (n strictly) in $\mathcal{X}$ that a hyperplane $\hat{\theta}$ in $\mathcal{Z}$ correspond to under the following cases:
check out the webcast for further details

(a) $\hat{\theta}_1 > 0, \hat{\theta}_2 < 0$.
When $\hat{\theta}_0 > 0$, the plot shows a "North-South opening hyperbola"; when $\hat{\theta}_0 < 0$, it's "East-West opening hyperbola" as shown in Figure 1.

Figure 1: Borders for the two cases of $\hat{\theta}_0$

→ For transformation of $\Phi(1, x_1, x_2)$ <-> $(1, x_1^2, x_2^2)$, if $\boldsymbol{\theta}_1 > 0$ and $\boldsymbol{\theta}_2 < 0$, if $\boldsymbol{\theta}_0$ (bias I think??) is > 0 we get the green left one if < 0 we get the blue one on the right

(b) $\hat{\theta}_1 > 0, \hat{\theta}_2 = 0$.
When $\hat{\theta}_0 > 0$, there's no border in $\mathcal{X}$ space, all data points $\mathbf{x}$ are classified into $y = +1$ category. When $\hat{\theta}_0 < 0$, the border is a single point at x_1 axis as shown in Figure 2.

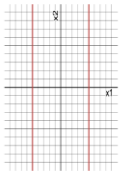


Figure 2: Borders when $\hat{\theta}_0 < 0$

(c) $\hat{\theta}_1 > 0, \hat{\theta}_2 > 0, \hat{\theta}_0 < 0$.
It's either a ellipse or a circle, as shown in Figure 3

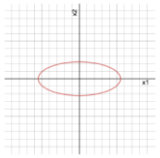


Figure 3: Borders when $\hat{\theta}_0 < 0$

(d) $\hat{\theta}_1 > 0, \hat{\theta}_2 > 0, \hat{\theta}_0 > 0$.
There's no border in $\mathcal{X}$ space. All data points $\mathbf{x}$ are classified into y well.

1. (MRQ with 4 options; 3 marks) Bias–Variance. Mark all statements that are true.
(a) A model with high variance will tend to have low test error over different samples of training data.
(b) An $\mathcal{H}$ with high complexity will tend to exhibit high variance.
(c) High variance models tend to have low bias.
(d) An $\mathcal{H}$ with low bias will generally improve over one with high bias, given sufficient training data.
Correct answers: (b). (c). (d)

Using a high variance vs low variance model when data has high stochastic noise

Explanation: Possible reasons to use a high variance model-
- When we have a large dataset, the errors seem to average out and we can learn an accurate representation
- If the training dataset is representative of the underlying target function, using a high variance model will be able to approximate the target quite accurately
- Useful for approximating high complexity target functions
- Bias-Variance tradeoff: Model with high variance will be helpful for finding a function with low bias (useful in cases stated in point 2).

Possible reasons for not using a high variance model-
- A high variance model will most likely lead to overfitting on the training dataset and not be able to generalise well.
- The model will closely capture the high stochastic noise

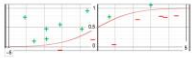
Why are linear models relevant in real world despite low expressive power compared to learners that capture both linear and non-linear relationships?

Why then are linear models still relevant in the real-world? Justify your response.

Explanation: Linear models are still relevant due to the following reasons. First, complex models (e.g., non-linear) may suffer from overfitting problems when the samples are insufficient. Second, it is easy to interpret and uses less computational resources. Lastly, linear models can represent non-linearity by using data transformation and kernel methods.

Logistic Regression is a classification algorithm! Based on our sigmoid curve, we get a probability, and based on our threshold, we output the classification y.

[Questions 1-2] The below image purportedly shows how a Logistic Regression classifer boundary looks like (A version of this image did actually appear on a fairly famous data science website).

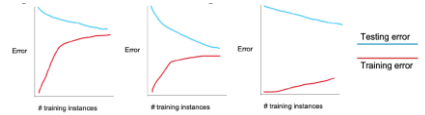


1. (Text Response; 3 marks) Describe why this is an incorrect representation of how Logistic Regression functions.
Explanation: There are several problems with the diagram as shown. For your understanding, we have made this model answer more comprehensive than is expected for full credit. In Logistic Regression, the red sigmoid-like curve is not a classification boundary, but rather a mapping function. Logistic Regression maps the real valued signal $\theta^T x$ to the bounded interval (0,1) as a probability of positive classification. As such, both positive and negative instances should not be depicted as being on the plane in the diagram itself.

2. (Text Response; 1 mark) Describe one aspect of this image which is representative of Logistic Regression.
Explanation: Instances lie directly on the red curve, such that their signal (as signified by the x value) maps directly to the probability of classification (the corresponding y value on the curve). The output of logistic regression is a classification, not a confidence value, so a more accurate diagram would have all of the the instances left of x=0 mapped to negative classifications (a row of instances along y=0) and those right of x=0 mapped to positive classifications (a row of instances along y=1).

Among analogizer, connectionist, symbolist and Bayesian algorithms, only connectionists use some form of randomization (SGD)

Graph of 3 models with different bias/variance



→ L = high bias (Error is high on both training and testing, i.e. H cannot model target well), variance is not high (testing/training error consistent)
→ M = moderate bias, moderate variance
→ R = Low bias (training error is low, so it can be modelled), high variance (testing error is very high), overfitted

We are learning a hypothesis function that approximates target function, NOT the other way round (below is false)

6. The goal of supervised learning is to learn a target function that approximates the hypothesis function.

→ We do not apply ML algorithms to outputs with 0 correlation, i.e. purely random (lottery, roulette)

Confusion Matrix

PREDICTED VALUES \ ACTUAL VALUES	POSITIVE	NEGATIVE
POSITIVE	TP	FP
NEGATIVE	FN	TN

Sensitivity / True Positive Rate / Recall

$$Sensitivity = \frac{TP}{TP + FN}$$

Specificity / True Negative Rate

$$Specificity = \frac{TN}{TN + FP}$$

False Negative Rate

$$FNR = \frac{FN}{TP + FN}$$

False Positive Rate

$$FPR = \frac{FP}{TN + FP} = 1 - Specificity$$

MAPE tutorial Question

$$MAPE = \frac{100\%}{n} \sum_{i=1}^{n} \left| \frac{y_i - \hat{y}_i}{y_i} \right|$$

(b) Assume that you are a supply-chain manager. You are using MAPE to judge your regression forecasts about **product demands** for the next month. We list them here. Discuss whether the listed MAPE shortcomings below will or will not affect you.
i. Data with zeroes or close to zeroes.
ii. Heavier penalty when predictions are higher than actual data.
iii. Assumption that zero in the data's unit of measurement holds meaning.

i. Yes, it will affect. Product demand can reach values of zero. MAPE produces undefined or infinite values when the actual values(y_i) are zero or close to zero.
ii. Yes, it affects as product demand cannot be negative values. MAPE is asymmetric when forecasts are strictly non-negative. It places a heavier penalty when forecasts($\hat{y}_i$) are higher than actuals(y_i). Percentage error cannot exceed 100% for low forecasts, but there are no upper limits for high forecasts as our product demands cannot be a negative value.
iii. No, this is not a shortcoming for us. MAPE assumes that when the value is zero, it is meaningful. In our case, forecasts for product demand being zero would cause MAPE to return 100% if our actual is non-zero. For units of measurements that have arbitrary zero values, using MAPE no longer makes sense.

4. **Data Resampling: SMOTE**
Refer the general Steps of SMOTE given to you below.
1. From all the data points of your minority class, pick a random point.
2. Find the k nearest neighbours to that point.
3. Pick one of the neighbours randomly, now we have a pair from the minority class.
4. Draw an imaginary line between the pair and pick a random point along the line.
5. The new random point is added to the minority class.

Figure 3 is a scatter plot of an imbalanced dataset to be used in a binary classification problem. Roughly illustrate how the transformed dataset will look like after SMOTE.
Taking k to be the default value of 5, we can find out all combinations of the minority class pairs. Add points along the lines connecting each minority class pair and we would get a result similar to what we see in Figure 4 after SMOTE. (We will get varying results as k changes)

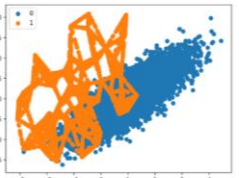
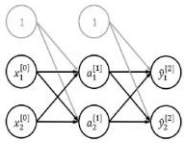


Figure 4: Scatter Plot after SMOTE transformation - Image Credits

→ Since we take the nearest neighbors to perform interpolation, we observe that the values has this 'star' like patten in this case, as new points can only be created along an imaginary line between those 2 points

1. **Backpropagation algorithm.** In this question, we're going to use a neural network with a **2-d input, one hidden layer with two neurons and two output neurons**. Additionally, the hidden neurons and the input will **include a bias**. We use **ReLU function** as the nonlinear activation function.
Here's the basic structure:



(a) Suppose there is a data input $\mathbf{x} = (2, 3)^\top$ and the actual output label is $\mathbf{y} = (0.1, 0.9)^\top$. The weights for the network are

$$W^{[1]} = \begin{bmatrix} 0.1 & 0.1 \\ -0.1 & 0.2 \\ 0.3 & -0.4 \end{bmatrix}, W^{[2]} = \begin{bmatrix} 0.1 & 0.1 \\ 0.5 & -0.6 \\ 0.7 & -0.8 \end{bmatrix}.$$

Calculate the following values after forward propagation:
$\mathbf{a}^{[1]}$, $\hat{\mathbf{y}}^{[2]}$ and $L(\hat{\mathbf{y}}^{[2]}, \mathbf{y})$.

$$\mathbf{a}^{[1]} = ReLU((\mathbf{W}^{[1]})^\top \mathbf{X})$$

$$\begin{aligned} a_1^{[1]} &= ReLU(x_0 \times W_{10}^{[1]} + x_1 \times W_{11}^{[1]} + x_2 \times W_{12}^{[1]}) \\ &= ReLU(0.1 + 2 \times (-0.1) + 3 \times 0.3) \\ &= ReLU(0.8) \\ &= 0.8 \end{aligned}$$

$$\begin{aligned} a_2^{[1]} &= ReLU(x_0 \times W_{20}^{[1]} + x_1 \times W_{21}^{[1]} + x_2 \times W_{22}^{[1]}) \\ &= ReLU(0.1 + 2 \times 0.2 + 3 \times (-0.4)) \\ &= ReLU(-0.7) \\ &= 0 \end{aligned}$$

$$\hat{\mathbf{y}}^{[2]} = ReLU((\mathbf{W}^{[2]})^\top \mathbf{a}^{[1]})$$

$$\begin{aligned} \hat{y}_1^{[2]} &= ReLU(a_0^{[1]} \times W_{10}^{[2]} + a_1^{[1]} \times W_{11}^{[2]} + a_2^{[1]} \times W_{12}^{[2]}) \\ &= ReLU(0.1 + 0.8 \times 0.5 + 0 \times 0.7) \\ &= ReLU(0.5) \\ &= 0.5 \end{aligned}$$

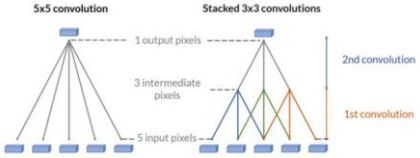
$$\begin{aligned} \hat{y}_2^{[2]} &= ReLU(a_1^{[0]} \times W_{20}^{[2]} + a_1^{[1]} \times W_{21}^{[2]} + a_2^{[1]} \times W_{22}^{[2]}) \\ &= ReLU(0.1 + 0.8 \times (-0.6) + 0 \times (-0.8)) \\ &= ReLU(-0.38) \\ &= 0 \end{aligned}$$

$$\begin{aligned} L(\hat{\mathbf{y}}^{[2]}, \mathbf{y}) &= \frac{1}{2} \times ((\hat{y}_1^{[2]} - y_1)^2 + (\hat{y}_2^{[2]} - y_2)^2) \\ &= \frac{1}{2} \times ((0.5 - 0.1)^2 + (0 - 0.9)^2) \\ &= \frac{1}{2} \times (0.16 + 0.81) \\ &= 0.485 \end{aligned}$$

→ Important: Remember to do ReLU after calculation!!! Or sigmoid or other actn fn

<u>CNN Design Choice (Stacking 2 conv layers of 3x3 vs Single conv layer of 5x5)</u>

→ 5x5 has 25 parameters, 2x3x3 has 18 parameters. With more parameters there is generally a higher chance of overfitting.
→ With 5 input pixels, 5x5 outputs 1 feature map, with 2x3x3, with 5 input pixels we get 5 → 3 → 1 input pixel as well, so the same amount of detail captured (collective receptive field) is the same. Take note this is assuming the stride of both convolutions is 1x1!!



5x5 convolution vs the equivalent stacked 3x3 convolutions

<u>More about RNNs</u>

2. **Quick Discussions.** Describe what deep learning models/techniques will you use to tackle the following problems.

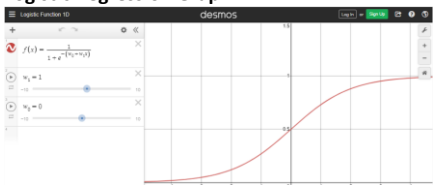
- You need a model to determine the sentiment of sentences.
RNN or CNN. A sentence has obvious sequential dependencies between words and hence a proper model would need to capture this interdependence. The Recursive neural network method is the standard method for dealing with any sequential (e.g., time series) input. RNNs natively deal with different length (sized) input (to think about, why?). As the final state of the RNN encodes the representation of the entire sentence, we use this final state to yield a classification of the sentiment of the sentence. This corresponds to a many-to-one RNN model. Also as the RNN tends to model recent past time steps well better than distant ones, several architectures improve on the plain RNN architecture; namely Long Short-term Memory (LSTM), Gated Recurrent Units (GRU), and their Bi-directional variants.
The CNN is also capable of doing sentence classification because the receptive fields of convolution layers are growing with the depth of the network. Deep layers are therefore able to capture the general context of the whole sentence. See Reading Material 1. CNN has a particular advantage in that the convolution operations can be done in parallel over a fixed-sized window.

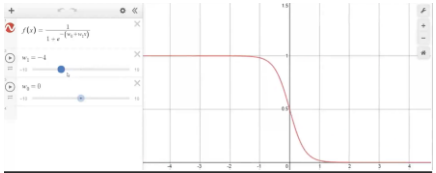
- You need a model to do translation between two languages.
RNN. This is a case of the many-to-many architecture, where the output comes strictly after encoding all of the input source sentence's token into a internal sentential representation (as evidenced by the final Θ_{hh} weights being passed after the final input token is observed. This sentence representation is sometimes referred to as a "thought vector". This encoded representation is then decoded to produce the target sentence in the output language. Nowadays, transformer models take the lead of area, and produce the SOTA result for many nlp problems i.e case2 and case3. Can refer to material 3 for more information.

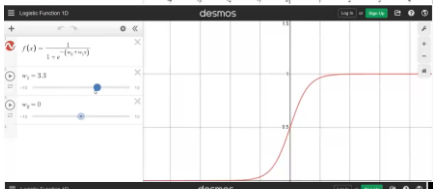
→ For many-to-many architecture, encode all of input source sentence's tokens first, then

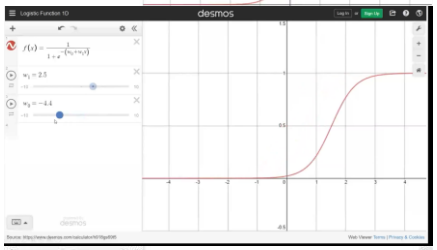
decoded to produce the target sentence in the output language

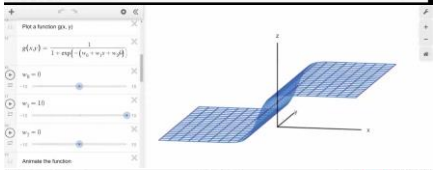
Logistic Regression Graph

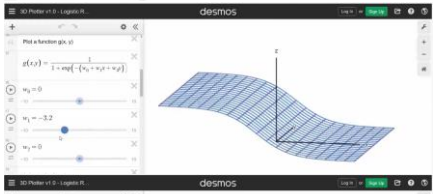


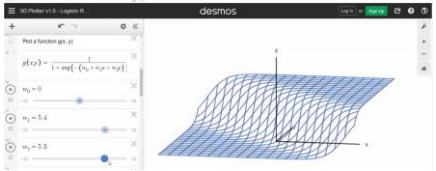


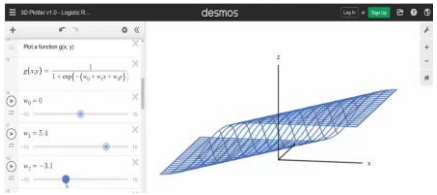












Backprop Chain Rule for RNN

1. **RNN and BPTT** Here, we'll be computing gradients via *Backpropagation Through Time* (BPTT). The Forward Pass of a RNN can be characterised as follows(Here σ denotes *softmax* function):

$$\mathbf{h}_t = g^{[h]}((\mathbf{W}^{[eh]})^\top \mathbf{x}_t + (\mathbf{W}^{[hh]})^\top \mathbf{h}_{t-1}) \quad (1)$$
$$\hat{\mathbf{y}}_t = g^{[y]}((\mathbf{W}^{[hy]})^\top \mathbf{h}_t) \quad (2)$$
$$\hat{\mathbf{o}}_t = \sigma(\hat{\mathbf{y}}_t) \quad (3)$$

The loss L is the Cross Entropy Loss:

$$L = -\sum_t^T \mathbf{y}_t \cdot \log(\hat{\mathbf{o}}_t) \quad (4)$$

For simplicity, let's call the final time step loss, $E_T = -\mathbf{y}_T \log(\hat{\mathbf{o}}_T)$. The objective of BPTT is to update the parameters $\mathbf{W}^{[eh]}$, $\mathbf{W}^{[hh]}$, and $\mathbf{W}^{[hy]}$.

(a) Use Chain Rule to find an expression for $\frac{\partial E_T}{\partial \mathbf{W}^{[hh]}}$. (Note, there is no need to expand the term $\frac{\partial \mathbf{h}_{T-1}}{\partial \mathbf{W}^{[hh]}}$ further)

The answer is a multiplication of the terms via Chain Rule:

$$\frac{\partial E_T}{\partial \mathbf{W}^{[hh]}} = \frac{\partial E_T}{\partial \hat{\mathbf{o}}_T} \times \frac{\partial \hat{\mathbf{o}}_T}{\partial \hat{\mathbf{y}}_T} \times \frac{\partial \hat{\mathbf{y}}_T}{\partial \mathbf{h}_T} \times \frac{\partial \mathbf{h}_T}{\partial \mathbf{W}^{[hh]}} \quad (5)$$
$$\frac{\partial E_T}{\partial \mathbf{W}^{[hh]}} = \frac{\partial E_T}{\partial \hat{\mathbf{o}}_T} \times \frac{\partial \hat{\mathbf{o}}_T}{\partial \hat{\mathbf{y}}_T} \times \frac{\partial \hat{\mathbf{y}}_T}{\partial \mathbf{h}_T} \times \left(\frac{\partial g^{[h]}(\mathbf{x}_T, \mathbf{h}_{T-1})}{\partial \mathbf{W}^{[hh]}} + \frac{\partial g^{[h]}(\mathbf{x}_T, \mathbf{h}_{T-1})}{\partial \mathbf{h}_{T-1}} \times \frac{\partial \mathbf{h}_{T-1}}{\partial \mathbf{W}^{[hh]}} \right) \quad (6)$$

Figure 1 shows the gradient flow.

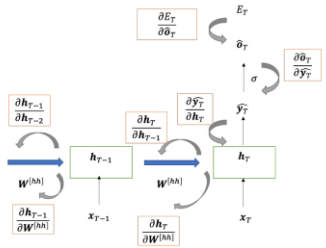


Figure 1: Gradient Flow

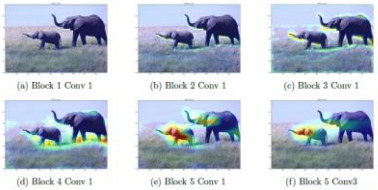
The term $\frac{\partial \mathbf{h}_T}{\partial \mathbf{W}^{[hh]}}$ is responsible for VGP. This is because the partial derivative of this term has an dependence on h_{T-1} from the previous timestep.

$$\frac{\partial \mathbf{h}_T}{\partial \mathbf{W}^{[hh]}} = \left(\frac{\partial g^{[h]}(\mathbf{x}_T, \mathbf{h}_{T-1})}{\partial \mathbf{W}^{[hh]}} + \frac{\partial g^{[h]}(\mathbf{x}_T, \mathbf{h}_{T-1})}{\partial \mathbf{h}_{T-1}} \times \frac{\partial \mathbf{h}_{T-1}}{\partial \mathbf{W}^{[hh]}} \right) \quad (7)$$

This happens all the way back in time till the first timestep. With every small number $0 < x < 1$ multiplied to the chain, the resulting gradient becomes even smaller. By the time the network tries to update the initial few layers (in time), the gradient would practically be 0. This means your network manages to update only the parameters from recent timesteps while ignoring the ones way back in time.

This further results in the *Short-term Dependency Problem* where the RNN's memory only goes back a few timesteps. It cannot keep track of things from the distant past.

Explaining Grad-CAM's saliency maps



(a) Block 1 Conv 1 (b) Block 2 Conv 1 (c) Block 3 Conv 1
(d) Block 4 Conv 1 (e) Block 5 Conv 1 (f) Block 5 Conv3

Figure 4: Heatmap through various Convolutional layers of VGG16 - Image Credit

(a) Comment on the heatmaps with respect to the image classification.

Despite being a classification network, VGG16 is able to locate the elephant in the picture. Moreover, the decisions are highly influenced by the face and ear portions of the data.

(b) In each layer, different segment of the image is activated. Why those activations are different? You can assume that *Block5 Conv3* is the decision making layer.

This is because earlier layers have filters which are looking at various parts of the image. But the class prediction largely depends on the activations of the layers just before prediction and hence, visualizing heatmaps of the layer just before prediction layer *Block5 Conv3* tells us which portion of the input image led to a particular prediction.

Minimizing loss on k-means clustering

(a) Show that each data assignment step (Line 5 in Algorithm 1) minimizes L_{clust}, given fixed cluster centers.

$$L_{clust} = \sum_{c=1}^{k} \sum_{x^{(j)} \in S_c} \|x^{(j)} - \mu_c\|^2 = \sum_{j=1}^{m} \|x^{(j)} - \mu_{assigned}\|^2 = \sum_{j=1}^{m} L_{x^{(j)}} \quad (2)$$

To minimize L_{clust}, we minimize the error for each data ($L_{x^{(j)}}$) independently since cluster centers are fixed. For each data, error is minimized if we assign it to the nearest cluster. Hence Line 5 minimizes L_{clust}.

(b) Show that each cluster center update step (Line 8 in Algorithm 1) minimizes L_{clust}, given fixed data assignments.

$$L_{clust} = \sum_{c=1}^{k} \sum_{x \in S_c} \|x - \mu_c\|^2 = \sum_{c=1}^{k} L_c \quad (3)$$

where $L_c = \sum_{x \in S_c} \|x - \mu_c\|^2$ represents the error for each cluster. Since assignments are fixed, we can optimize for each cluster independently. To minimize L_c, we update the cluster centers to the mean of all data assigned to the cluster. This is very intuitive but can be proven using simple math.

$$L_c = \sum_{x \in S_c} \|x - \mu_c\|^2 = \sum_{x \in S_c} \sum_{d} (x_d - \mu_{c,d})^2 \quad (4)$$

In the above equation, $x = \{x_1, x_2, ..., x_n\}$ and $\mu_c = \{\mu_{c,1}, \mu_{c,2}, ..., \mu_{c,n}\}$. Hence, $\mu_{c,d}$ indicates d-th element of vector μ_c. L_c is minimized when derivatives with respect to $\mu_{c,d}$ equals zero. Let's calculate the derivative first.

$$\frac{\partial L_c}{\partial \mu_{c,d}} = \sum_{x \in S_c} \frac{\partial}{\partial \mu_{c,d}} (x_d - \mu_{c,d})^2 = \sum_{x \in S_c} -2(x_d - \mu_{c,d}) \quad (5)$$

If L_c is minimized, this derivative equates to zero.

$$\frac{\partial L_c}{\partial \mu_{c,d}} = 0$$
$$\sum_{x \in S_c} (x_d - \mu_{c,d}) = 0$$
$$\left(\sum_{x \in S_c} x_d \right) - m_c \mu_{c,d} = 0$$
$$\mu_{c,d} = \frac{1}{m_c} \sum_{x \in S_c} x_d \quad (6)$$

where m_c is the number of data being assigned to the cluster c. It can be further shown using second derivatives that this is a minimum point.

Overcoming sub-optimal clustering in k-means clustering (k-means++)

- *The performance of k-Means is sensitive to the centroid initialization, which can be improved by k-Means++. k-Means++ is shown below (credit to Wikipedia, not required for examination).*

i. *Choose one centroid uniformly at random among the data points.*
ii. *For each data point x not chosen yet, compute $D(x)$, the distance between x and the nearest existing centroid.*
iii. *Choose one new data point at random as a new centroid, using a weighted probability distribution where a point x is chosen with probability proportional to $D(x)^2$.*
iv. *Repeat Steps ii and iii until k centroids have been chosen.*
v. *Now the initial centroids are chosen, proceed using standard k-means clustering.*

Disadvantages of using auto-encoder for image compression

(b) Autoencoder can compress the original input into a lower dimensional encoding. However, it is hardly used in practice for image compression. List the disadvantages of autoencoder when used for compression.

Lossy. Autoencoder loses pixel information because it aims to discard the unimportant features and retain only important features. On the contrary, modern image format like PNG can achieve lossless compression.
Data Dependent. Autoencoder works well only if the input comes from the same distribution of the training set. If the input is out of distribution, the recovered image quality will be very poor.
Despite the disadvantages, autoencoder is an important model for being unsupervised. Unsupervised learning is an important research field because it enables machine learning models to leverage the large amount of cheap unlabeled data on internet. The idea of reconstructing the input is further developed and inspired the SOTA unsupervised deep learning models, including GPT (language) and CLIP (vision + language).

→ Also, the latent vector produced is not always the same whenever we train a model, so it may not produce a consistent way of decoding an image after encoding it

TBPTT (Truncated BPTT)

- Overcomes vanishing gradient problem
- Considers a moving window through the training process
→ Reduces training time, since we only run bptt on a chunk (window) at a time, and less complex
→ Dependencies longer than chunk length are not taught
→ stop after a set number of backpropagation steps or till we have fully backpropagated the loss through an input instance. Latter happens when the sequence is longer than the threshold set of the BTT truncation
→ processes the sequence one timestep at a time, and every k1 timesteps, it runs BPTT for k2 timesteps

The TBPTT algorithm requires the consideration of two parameters:

- **k1**: The number of forward-pass timesteps between updates. Generally, this influences how slow or fast training will be, given how often weight updates are performed.
- **k2**: The number of timesteps to which to apply BPTT. Generally, it should be large enough to capture the temporal structure in the problem for the network to learn. Too large a value results in vanishing gradients.

Precision / Recall / Accuracy when to use

Q1) [4 Marks] MRQ: What metric(s), which is not misleading, should you use to report how well the model is doing?

a) Cosine Distance
b) Accuracy
c) Recall of Faulty prediction
d) Precision of Faulty prediction

Answer: c

Justification / Working:

a) False. Cosine similarity is measured for vector regression.
b) False. Since the data is imbalanced, accuracy is misleading.
c) True. Car engine diagnosis is similar to rare disease diagnosis. Missing to diagnose problems may cause accidents, hence false negative is very costly. Recall $R = TP/(TP + FN)$ is more important to decrease.
d) True. Prediction $P = TP/(TP + FP)$. But False Positive (FP) is less important, since this just means some wasted work.

- Use accuracy when data is balanced
- If its very costly to have false negatives (engine breakdown, cancer prediction), check recall
- If its very costly to have false positives (criminal sentencing?), check precision