1 Model Evaluation

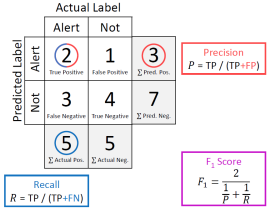
1.1 Classification Metrics

1.1.1 Accuracy

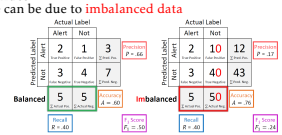
$$A = \frac{1}{m}\sum_{j=1}^{m}[\hat{y}_j = y_j]$$

- $\hat{y}_j$ is the predicted value
- y_j is the ground truth value
- Interpreted as ratio of predicted elements that are equal to ground truth over total elements
- Accuracy other formula: $\frac{TP+TN}{TP+TN+FP+FN}$

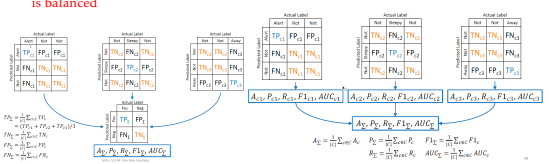
1.1.2 Confusion Matrix



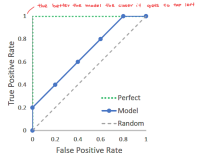
- Maximize **recall** to minimize **false negatives**
- Maximize **precision** to minimize **false positive**
- Only use F1 score if you cant justify between recall and precision
  - F1 score is not just a simple average but is instead more robust as it is **less sensitive to extreme values**
- Low metric score can be due to imbalanced data



- Depends on **prediction threshold**
- If there is multiclass, we can use either **micro-average** or **macro-average**
  - Micro-average weighs each **instance** equally and accounts for imbalanced data
  - Macro-average weight each **class** equally and is used only when real-world test data is balanced

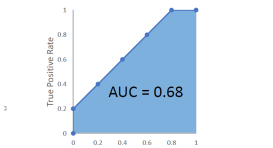


1.1.3 Receiver Operator Curve



- Diagonal random line indicates 50% chance of correctness.
- If ROC curve is above the random line, model is more accurate than chance.
- Perfect curve has TPR = 1 and FPR = 0 always.

1.1.4 Area Under Curve (AUC) of ROC



- AUC is a concise **metric** instead of a full figure.
- Concise metrics enable *clearer comparisons*.
- **AUC > 0.5** means the model is better than chance.
- AUC ≈ 1 means model is very accurate.

1.2 Regression Evaluation Metrics

1.2.1 Average Difference Metrics

Mean Absolute Error (MAE)

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

Mean Squared Error (MSE)

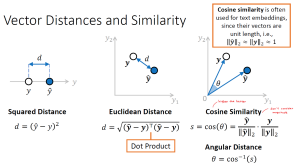
$$MSE = \frac{1}{m}\sum_{j=1}^{m}(\hat{y}^{(j)} - y^{(j)})^2$$

Root Mean Squared Error (RMSE)

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

MSE and RMSE penalize larger differences more than MAE

1.2.2 Vector Distances and Similarity



1.2.3 Difference between MSE and Euclidean Distance

- MSE is the average of all the distances across the **test set** whereas euclidean distance is calculating the distance **across feature vector**

- Euclidean distance $d_j = \sqrt{(\hat{y}_j - y_j)^\top(\hat{y}_j - y_j)} = \sqrt{\sum_{i=1}^{n}(\hat{y}_i^{(j)} - y_i^{(j)})^2}$
- $MSE = \frac{1}{m}\sum_{j=1}^{m}\sum_{i=1}^{n}(\hat{y}_i^{(j)} - y_i^{(j)})^2$

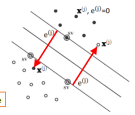
2 Data Processing

2.1 Linear Separability

- In most real-world scenario, **data tends to not be linearly separable** (e.g. images, time-series data, text). Wrongly assuming that data features are linearly separable leads to irrelevant features being **uninformative** to train the model to discriminate between prediction labels
- Check using **scatterplot** (if only 2 features) and **scatterplot matrix** (if more than 2 features). We can also obtain a computational metric using **linear SVM**

2.1.1 Testing Liner Separability using Linear SVM

- Each $e^{(i)}$ is the **distance** that the misclassified point i is from its correct margin
- Total violation: $\sum_{i=1}^{m} e^{(i)}$
- Calculating the **total violation** indicates how **linearly separable** the data is in terms of its features
- Higher violation => Less linearly separable



2.1.2 Mitigation

1. Find useful features through feature extraction (collecting new features)
2. Transform features
   - Feature engineering ($x \rightarrow x^2$)
   - Change Basis Vectors (PCA, LDA)
   - Kernel trick
   - Features Learning (Neural Network)

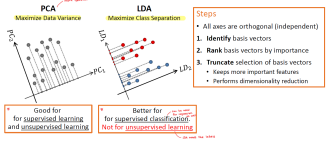
2.1.3 Principal Component Analysis (PCA)

- PCA uses orthogonal transformation to convert a set of observations into a set of values of linearly uncorrelated variables called principal components
- Each principle component is a mix of features (e.g. PC1 can be 4 parts x_1, 1 part x_2)
- We can then plot out something called a scree graph which tells you which Principle components are important and we can just keep those to **reduce dimensionality**
- Higher variance means more information
- **Feature Normalization** needs to be done

2.1.4 Linear Discriminant Analysis (LDA)

- LDA works by projecting the data onto a lower-dimensional space that maximizes the separation between the classes
- Uses the F-test metrics: $F = \frac{|\mu_r - \mu_b|^2}{s_r^2 + s_b^2}$
- The F-test scores are tied to each linear discriminant and the **higher F-test score means it separates classes better**
- Discard LD with low F-test to reduce dimensions

2.1.5 PCA vs LDA



2.2 Curse of Dimensionality

- Curse of dimensionality occurs when there are way more features than data points ($n \gtrsim m$) are if we are dealing with unstructured data (e.g. images, sensor data)
- Causes problems like **overfitting** since data are too sparse to inform about true decision boundary and model can easily fit to sparse data
- Also a problem for clustering (e.g. knn) since distance are too similar
- Check using histogram of distances (check for **variance**)
  - Generally tedious to check various, general rule of thumb is for features to be $\frac{1}{5}$ of data points

2.2.1 Migrations

- Feature selection
  - Wrapper method (e.g. Recursive Feature Elimination, **Elbow method**)
  - Filter method (e.g. Only keep high info gain features, remove correlated features)
- Dimensionality Reduction
  - Linear Matrix Factorization: PCA, LDA
  - Auto Encoders
  - Feature Selection helps to enable **faster model training** and **better intepretability**

2.3 Imbalanced Data

- Values not evenly distributed in features due to rare events (e.g. rare cancer) or skewed data collection
- Leads to evaluation metrics being **misleading** to interpret and models **overfitting** to majority class
- Can be checked easily by using histogram or bar chart of features

2.3.1 Mitigation

- Collect more data instance
- Resample instances (undersampling, oversampling, SMOTE)
- Undersampling should only be used when there is not much loss of data (e.g. 55% majority class VS 45% minority class)
- If data is very largely imbalanced, then oversampling/SMOTE is preferred so that we don't lose too much data

3 Feature Engineering

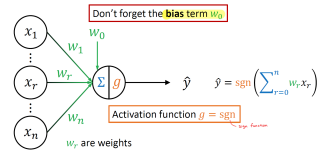
3.1 Feature Extraction and Engineering

- Process of **transforming raw data** to improve the accuracy of models
- Benefits:
  - **Capture domain knowledge** (e.g., periodicity or relationships between features)
  - **Express non-linear relationships** using linear models
  - **Encode non-numeric features** to be used as inputs to models

Tabular	Temporal	Image	Text
• Domain-specific custom equations • Features from counting, aggregation, difference, min, max	• Features from previous values, aggregate statistics, linear regression • Wave analysis features	• RGB image as 3D tensor • Color features from RGB histogram • Shape features from edge detection • Edge detection via Convolution	• Tokenization • Stemming, Lemmatization • Stop words • Bag-of-Words encoding

4 Perceptron & Neural Networks

4.1 Perceptron Classification



- Takes in a vector of input which are assigned **weights**
- Prediction class $\hat{y}$ is given by applying the **activation function** g on the summation of the individual inputs multiplied by their weights $\rightarrow \hat{y} = g(w \cdot x) = g(w^T x)$

4.2 Perceptron Learning Algorithm (PLA)

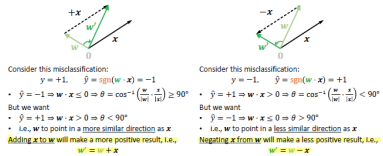
- One of the simplest way to learn what weights to assign to inputs and is **typically not used** in modern ML approach. Never converges for non-linear or noisy data

1. Initialize weights $\boldsymbol{w}$
   - Could be all zero, or random small values
2. For each instance i with features $\boldsymbol{x}^{(i)}$
   - Classify $\hat{y}^{(i)} = \text{sgn}(\boldsymbol{w}^T\boldsymbol{x}^{(i)})$
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

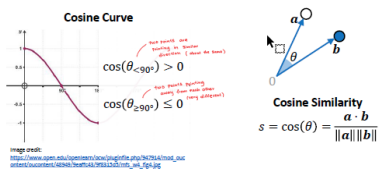
$$\begin{pmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_n \end{pmatrix} \leftarrow \begin{pmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_n \end{pmatrix} + \eta(y - \hat{y})\begin{pmatrix} 1 \\ x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

$$w_r \leftarrow w_r + \eta(y - \hat{y})x_r$$

- The update of weight is given by $w \leftarrow w + \eta(y - \hat{y})x$, where LHS w is the new weight, RHS w is the old weight, η is the learning rate and $(y - \hat{y})$ is the learning error



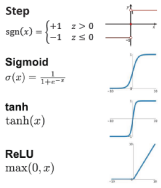
- Metric used to measure learning error is **cosine similarity**



4.3 Perceptron vs Linear SVM

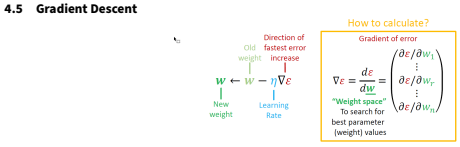
- Can select any model to linear => not robust (learns different weights for different initializations)
- Cannot converge on non-linearly separable data
- Perceptron of "optimal stability"
- Maximizes margin
- Soft-margin: allows soft error => can learn from non-linearity separable data

4.4 Activation Functions



- 4 common activation functions are sign, sigmoid, tanh and ReLU
- All 4 activation functions are **differentiable**, allowing for gradient descent to be used to find minimum of error faster
- Sigmoid, tanh and ReLU are non-linear functions which allows for models to introduce **non-linearity** and create a non-linear, more complex decision boundary

4.5 Gradient Descent



Binary Cross-Entropy error (for classification)

$$\varepsilon = -[y \log \hat{y} + \neg y \log \neg \hat{y}]$$

Square Error (for regression)

$$\varepsilon = \frac{1}{2}(y - \hat{y})^2$$

4.5.1 Gradient of Error

- Uses chain rule to determine the gradient/derivative of error

$$\varepsilon(w) = \varepsilon\Big(g\big(f(w)\big)\Big)$$

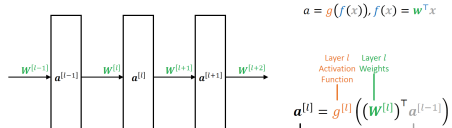
$$\varepsilon = \varepsilon(g), \quad \hat{y} = g, \quad g = g(f), \quad f = w^\top x$$

$$\nabla_w \varepsilon = \frac{d\varepsilon}{dw} = \frac{d\varepsilon}{dg}\frac{dg}{df}x$$

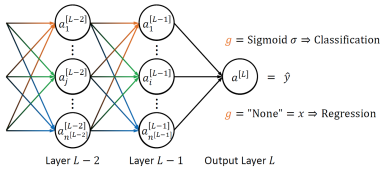
4.5.2 Summary of Gradient Calculation for Activation Functions

Error $\varepsilon(\hat{y})$	$\frac{d\varepsilon}{d\hat{y}}$	Activation $g(f)$	$\frac{dg}{df}$	Weighted Sum $f(w)$	$\frac{df}{dw}$
Square Error	$\frac{1}{2}(y-\hat{y})^2$	$-(y-\hat{y})$	Sigmoid	$\frac{1}{1+e^{-f}}$	$(1-g)g$	$w^\top x$	x
Binary Cross Entropy	$-y\log\hat{y}$ $-\neg y\log\neg\hat{y}$	$-\frac{y}{\hat{y}}+\frac{\neg y}{\neg\hat{y}}$	tanh	$\tanh f$	$1-g^2$		
Error $\varepsilon(x)$	$\frac{d\varepsilon}{dx}$	ReLU	$\max(0,f)$	$[f>0]$			
Categorical Cross Entropy	$-y\cdot\log\hat{y}$	$\hat{y}-y$	Softmax	$\frac{e^f}{e^f\cdot\mathbf{1}}$	$\frac{dg}{dx}$	Advanced: further reading	

4.6 Artificial Neural Network (ANN)

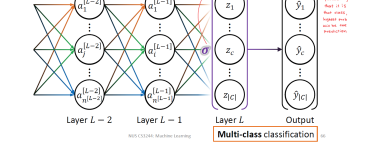


4.6.1 Single-output Neural Network



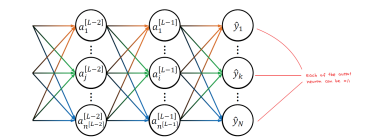
4.6.2 Multi-class Neural Network

Error $\varepsilon = -\boldsymbol{y} \cdot \log \hat{\boldsymbol{y}}$

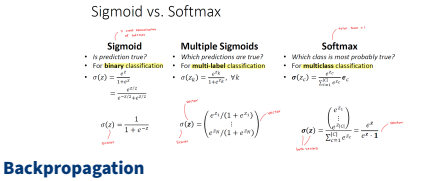


4.6.3 Multi-label Neural Network

Error $\varepsilon = \|y - \hat{y}\|^2$



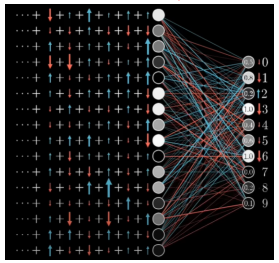
4.6.4 Sigmoid vs Softmax



5 Backpropagation

5.1 What is Backpropagation

- Used to **efficiently** compute the **gradient of the weighted input of each layer** from the last layer of the neural network to the front
  - avoids duplicate calculation
  - doesn't compute unnecessary intermediate values
  - computes gradient of each layer
- Idea for backpropagation is that the error of the prediction is propagated backwards through network, leading to adjustment of the weights and biases to minimize error
- Backprop relies on the fact that each function/layer is differentiable



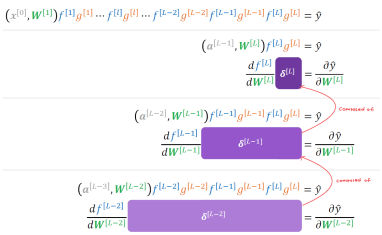
5.2 Important Math Notations

$$\frac{d\hat{y}}{d\mathbf{W}^{[l]}} = \boldsymbol{a}^{[l-1]}(\boldsymbol{\delta}^{[l]})^T$$

$$\boldsymbol{\delta}^{[l]} = \left(\frac{dg^{[l]}}{df^{[l]}}\right)\left(\mathbf{W}^{[l+1]}\boldsymbol{\delta}^{[l+1]}\right)$$

$$\frac{d\varepsilon}{d\mathbf{W}^{[l]}} = \frac{d\varepsilon}{d\hat{y}}\frac{d\hat{y}}{d\mathbf{W}^{[l]}}$$

5.3 Backpropagation Visualization



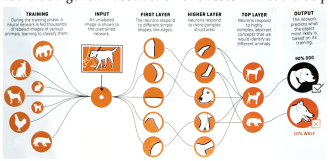
6 Convolutional Neural Network

6.1 Deep Neural Network

- Basically a neural net **that has ≤ 3 hidden layers**
- Layerscan be convolution, pooling, concatenation, dropout, fully connected or softmax
- Use to **predict complex target functions** in the real world with **a lot of parameters**
- Requires large amount of data to increase performance and avoid curse of dimensionality

6.2 Convolutional Neural Network

- Suitable for image processing
  - Image classification - detect face emotions
  - Object detection - self-driving cars
  - Image segmentation - cancer cell detection
- Convolutionary Neural Networks layers are analogous to **applying different filters and kernels** where it gradually builds up in complexity
  - First layer could be simple edge/shape detection
  - Higher level can have more complex abstractions like shape of a tail, shape of a leg
  - Top layer would be a general abstraction of classes that could be predicted



6.3 Kernel Size, Stride, Padding

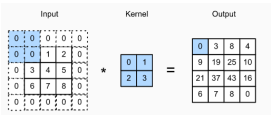


Figure 1: Padding

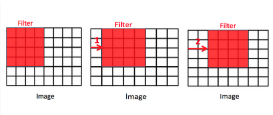


Figure 2: Stride

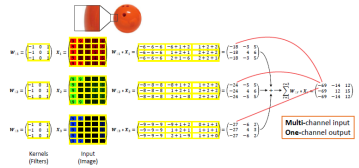
6.3.1 Calculation for Output Dimension

$$\dim \boldsymbol{y} = \left\{\left(\left\lfloor \frac{h_x + h_p - h_k}{h_s} \right\rfloor + 1\right) \times \left(\left\lfloor \frac{w_x + w_p - w_k}{w_s} \right\rfloor + 1\right)\right\}$$

- h_x, w_x is the row x col of the input
- h_p, w_p is row x col of padding
- h_k, w_k is row x col of kernel
- h_s, w_s is row x col of stride

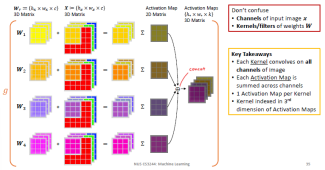
6.4 Multi-Channel Convolution

- Have 1 kernel for each channel (R, G, B) of the image, do convolution and then add up the resulting 2d matrix

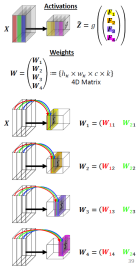


6.5 Multiple Convolution Kernels

- Each kernel now convolves on **all channels** of the image
- Can think of **each kernel looking for specific "feature"** of the image (e.g. which part was more yellow, pink, orange etc.)
- Results in an **activation map** that basically tells you which part of the image each feature lies more in
- Concat the activation maps together in final output

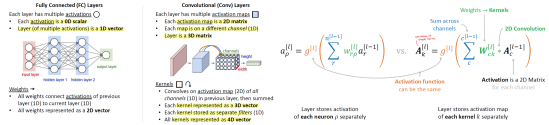


6.6 Multi-Channel Multi-Kernel Convolution



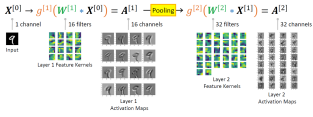
- Intuition is that multi-channel is used to detect color-specific features whereas multi-kernel can be used to detect different kinds of features (e.g. edges, textures etc.)
- In a multi-channel multi-kernel convolution operation, **each kernel is applied to the corresponding channel** of the input data, and the results are summed to produce the final output feature map. This process allows the network to **detect complex patterns** in the input data by **combining the features detected by the different kernels**

6.7 Fully Connected vs Convolution Layers



6.8 Convolutional Layers

- Typically **starts off with relatively low number of kernels** to detect basic features, **increase number of kernels in deeper layers** to get **more descriptive power** as we combine more and more simple features



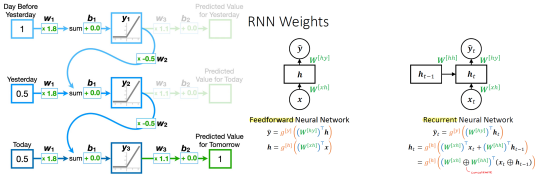
- Uses **pooling** layers to **downsample** feature maps which helps to train later kernels to detect **higher-level** features
- Also helps in **reducing dimensionality**
- Typical methods are max-pool (most common), average pool, sum-pool

7 Recurrent Neural Network

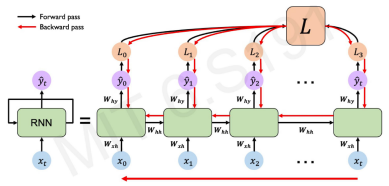
- Suitable for **text** and **time series** data
- Used in applications like speech recognition, music generation, sentiment classification, DNA sequence analysis, machine translation
- Output of the hidden layers of the **previous time step** is fed into the training of the hidden layer of the **next time step**
  - Notion of **using past data to predict things in the future**
  - We are using the output of the hidden layer of the previous time step and not just the input data of the previous time step as hidden layer contains a **more "compressed" representation** of the input data with **mostly useful information**
- Weights for training each time step remains the same for **efficiency** and to ensure that model can **predict independent of time**

$$\hat{y}_t = g^{[y]}(h_t)$$

$$h_t = g^{[h]}(x_t, h_{t-1})$$

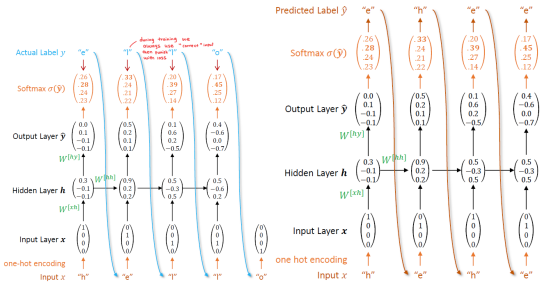


7.1 Backpropagation Through Time



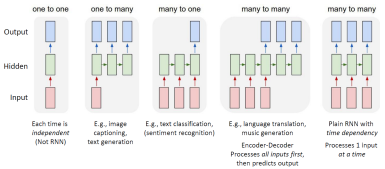
- In typical feed-forward neural nets, the backpropagation only occurs through **one of the verticals in the diagram**
- However, in RNN, since there is a notion of time data, we also have to propagate the weight information through the **horizontal (time) axis**
- **Extremely slow** since we have to complete the entire sequence for all the time steps before we can do any gradient/weight updates
  - **Transformers** are a way to speed them up by calculating the backprop in parallel without needing to depend on the next time step

7.2 Training RNN vs Prediction using RNN



- During training, we will always want to feed in the **actual label as input** for each of the time step
- During prediction, the input of the next timestep will be the **predicted label** in the previous timestep

7.3 Types of Sequence Modeling

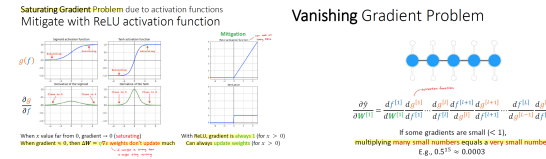


7.4 RNN Training Issues

7.4.1 Overfitting

- Recap: overfitting is when the performance of validation < training
- RNN/Deep Learning models can overfit when there are **too many parameters** and model is **more expressive than needed**
- Mitigation:
  - Gather more data and get larger training dataset
  - Implement a **dropout** layer - **randomly drop out** neurons during batch training so that you **disallow propagation** through these neurons
    - Note that **all neurons still used for prediction**

7.4.2 Saturating/Vanishing Gradient Problem



8 Explainable AI (XAI)

8.1 Feature Importance

- Want to explain what features are **important** for prediction and in what ways the features **influenced** the prediction
- Implementations:
  1. **Weights** for linear/logistic regression
  2. **Surrogate Weights** from LIME
  3. **Saliency Maps** (heatmap) of CNN

8.2 Intepreting Logistic and Linear Regression

Figure 3: Linear Regression

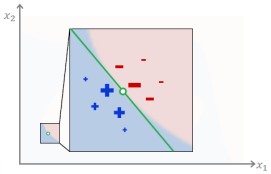
Figure 4: Logistic Regression

8.3 Local Interpretable Model-Agnostic Explanations (LIME)

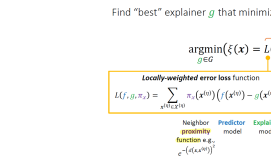
8.3.1 Naive Approaches to Describe Non-linear Decision Boundary

1. Use **linear model** to explain a complicated non-linear model
   - Simple to interpret
   - Too much error between explanation function and actual prediction function - lots of false positives and false negatives if we use simple linear model
2. Use gradients to explain non-linear model
   - Calculates steepness for each feature x_r
   - Difficult to remember, since gradients are different for each instance (point)

8.3.2 LIME



1. Choose an instance $\boldsymbol{x}$ to explain
2. Focus on a local region
3. Get training set as the neighbors of $\boldsymbol{x}$
4. Train a surrogate **linear** model $g(x) = w \cdot x = \sum_{r=0}^{n} w_r x_r$, where the weights are higher the closer they are to x



- Model-agnostic means that LIME can be used for explaining any models
- Note that it only gives explanation for a local area
  - If area is defined big enough then can explain "globally"

8.4 Gradient-Weighted Class Activation Mapping (GRAD-CAM)

- Used when explaining **images** as it has **too many features** which is unsuitable for LIME
- Explains using something called **saliency map** which is basically a heat map
- Grad-CAM aims to apply to idea of feature importance to activation maps

1. Compute Activation Maps $\boldsymbol{A}^{[L]}$ 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: $\bar{y} = \begin{pmatrix} \hat{y}^{(1)} \\ \hat{y}^{(2)} \\ \hat{y}^{(3)} \\ \hat{y}^{(n)} \end{pmatrix}, \boldsymbol{e}^{(c)} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}$, then $\bar{y}^{(c)} = \bar{y} \circ \boldsymbol{e}^{(c)} = \begin{pmatrix} 0 \\ 0 \\ y^{c} \\ 0 \end{pmatrix}$
   2. To generate explanation only for that class c
4. Compute importance weight $\alpha_k^{(c)}$ for each Activation Map $\boldsymbol{A}_k^{[L]}$
   1. Backprop from $\bar{y}^{(c)}$ to get gradients (relative to activations) at last conv layer
5. Compute weighted sum with ReLU to get **Class Activation Map**

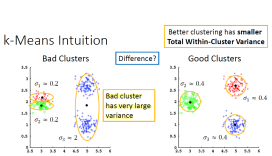
9 Unsupervised Learning

9.1 K-Means Clustering

9.1.1 Clustering Applications

- Customer segmentation, Image Segmentation, Behavior Segmentation, Color Quantization (reduce image size)

9.1.2 Intuition of K-Means

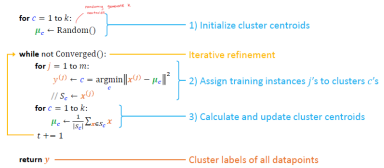


- Group clusters and try to **minimize** within-cluster variance
- More variance $\rightarrow$ more dissimilarity between members of the same cluster
- Metric for variance is **Within-Cluster Sum-of-Squares** (WCSS) L:

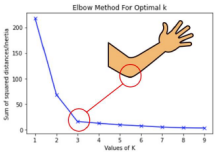
$$\underset{s}{\operatorname{argmin}} \sum_{c=1}^{k} \sum_{x \in S_c} \|x - \mu_c\|^2$$

- WCSS is the sum of the squared distances between each data point in a cluster and its centroid. It's calculated for each cluster and then summed up for all clusters.

9.1.3 k-means Clustering Algorithm



9.1.4 Elbow Method



- Used to determine k (number of clusters)
- k increases with diminishing returns $\rightarrow$ when k is too high, there is a marginal decrease in WCSS
- Ideal k is the points just before the marginal returns starts

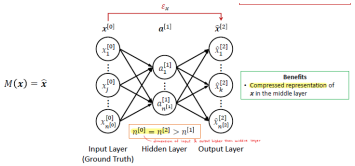
9.1.5 k-means VS kNN

	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

9.2 Autoencoders

- Autoencoders are designed to efficiently **compress and encode** data
- Primarily used for **dimensionality reduction** or **feature learning**

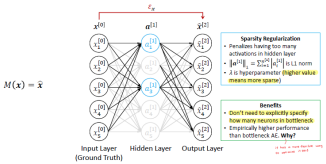
9.2.1 Auto-Encoder with bottleneck



- Hidden layer contains a compressed representation of x which has **less redundant information**
- Used to **learn important features** and **reduce dimensionality**
- Uses the reconstruction loss function:

$$\varepsilon_x(\hat{x}, x) = (\hat{x}, x)^T (\hat{x}, x)$$

9.2.2 Sparse Autoencoder



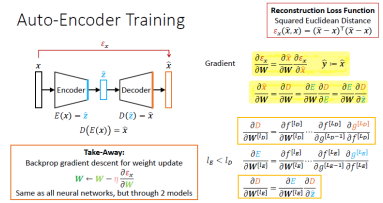
- Uses sparsity regularization to **penalize too many activations** in hidden layer
- Adds a L1 norm term to regular reconstruction loss function: $\lambda \left\| a^{[1]} \right\|_1$, where higher λ means more sparse
- **Don't need to explicitly specify** number of neurons in bottleneck and have empirically higher performance due to it being **more flexible**

9.3 Comparison with PCA

- Both have ability to reduce dimensionality
- Autoencoders with linear activation**linear activation** is similar to PCA but with non-orthogonal latent features
- Autoencoders performs better than PCA with **non-linear activation** since it can be more expressive

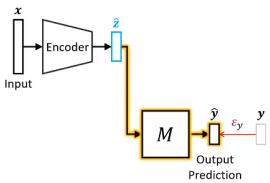
9.4 How to train Autoencoders

- Basically the same as normal neural networks using **backpropagation** with gradient descent but through 2 **models** (encoder and decoder)



9.5 Applications of Autoencoder

9.5.1 Feature Representation Learning



- Basic idea is that we want to use Autoencoder to do feature learning in an **unsupervised** way to get some form of compressed latent feature representation
- We then freeze encoder layer and **discard decoder layer** and instead train a Fully Connected neural network using this latent representation (efficient since we training on compressed, useful information only)
- Benefit is that you **can train on a lot of unsupervised data** and test and tune using just a small subset of test data

9.5.2 Anomaly Detection

- Idea is to train autoencoder which is very good at reconstruction **regular** instances
  - Only train on **"good"**instances
- When an irregular input comes in and autoencoder tries to reconstruct it, **error rate will be high** and we can catch it

9.5.3 Denoising Autoencoder for Robust Learning

- Idea is that you perturb dataset with noise and pass it to autoencoder to train
- Since autoencoders are good at removing redundant information, the goal is that autoencoder will only pick up on the non-noisy part of data and **ignore noise**
- Helps to get models which are **resistant and robust to noise**