Decision Stumps and Trees

- A decision stump is a single decision model that trains to find an optimal feature and threshold to split. if $x_{ij} > t$ then $y_i = 1$ otherwise $y_i = 0$. Stump model trains on data by by finding best feature and threshold for minimizing error or maximizing information gain.
- A decision tree model is constructed with decision tree stumps, using recursive greedy splitting. This is where stumps are created using the data set after a split to further split on the data.
- Advantages: easy to implement, easy to understand, and fast to learn.
- Disadvantages: Greedy recursion construction makes it difficult to find optimal rules, leading to inaccuracies.
- Model complexity increases with tree depth which can lead to overfitting.

Error $= \frac{1}{n}\sum_{i=1}^{n}\mathbb{I}(y_i \neq \hat{y}_i)$
Information Gain $=$ entropy$(y) - \frac{n_{yes}}{n}$entropy$(y_{yes}) - \frac{n_{no}}{n}$entropy$(y_{no})$
entropy(y) $= \sum_{c=1}^{k}\frac{n_c}{n}\log(\frac{n_c}{n})$ k is the number of unique labels in y
$O_{train}(ndth)$, $O_{test}(th)$
t = number of thresholds, h = depth of a tree (hyper parameter)

Naive Bayes Classifier

- A Naive Bayes classifier is a probabilistic model based on applying Bayes' theorem with the assumption that the features are conditionally independent given the class label.
- The model predicts the class label y_i for an input x_i by calculating the posterior probability $P(y_i|x_i)$ using:

$P(y_i|x_i) = \arg\max_{c\in K} P(y_c)\prod_{j=1}^{m} P(x_{ij}|y_c)$ *K is the set of class labels

- For training the likelihood $P(y_c)$ is computed for each label and $P(x_{ij}|y_c)$ is computed for each feature given each label
-to deal with some $P(x_{ij}|y_c)$ probabilities equalling zero, Laplace smoothing is used to make these probabilities non-zero:

$P(x_{ij}|y_c) = \frac{count(x_{ij},y_x)+\beta}{count(y_c)+\beta k}$

- as β increase, the conditional probabilities become more uniform
- Advantages: Works well with small datasets, fast to train and predict, easy to implement, and handles high-dimensional data well.
- Disadvantages: Strong independence assumption is rarely true in real-world data, leading to suboptimal results if features are highly correlated.
- The Naive Bayes classifier can be used for binary and multi-class classification problems.

$O_{train}(ndk)$, $O_{test}(tdk)$
n = number of samples, d = number of features, k = number of class labels

---

K-Nearest Neighbors Classifier

- K-Nearest Neighbors (KNN) is a non-parametric, instance-based learning algorithm that classifies a sample based on the majority class among its k nearest neighbors in the feature space.
- k is a hyper-parameter
- The model predicts the class label y_i for an input x_i by finding the k nearest neighbors (based on a distance metric, typically Euclidean distance) and performing majority voting.
- The value of k is a hyperparameter that controls the number of neighbors used in the voting process. Smaller k makes the classifier sensitive to noise, while larger k smooths the decision boundaries. E_{train} when k = 1, is going to be zero.
- Advantages: Simple to implement, no training phase, effective in low-dimensional data, non-parametric (no assumptions about data distribution).
- Disadvantages: Computationally expensive at test time (due to distance calculations), sensitive to irrelevant or redundant features, requires careful tuning of k and distance metric.
- $O_{train}(1)$ (since there's no explicit training), $O_{test}(ndt)$

Random Forest

- An ensemble method where multiple decision tress are made and used to assess the model to make predictions. Consensus method van vary but most common is voting on most common label the trees predict for each example. Random forest has higher accuracy compared to single decision tree.
- during training, training data is bootstrapped for each decision tree and each tree is given a randomize subset of features to train on.
- - Boot strapping is the method of generating a new training data set from the original data set by randomly copying examples from the original data and adding them to the new data set. A new data set is produced for each tree to train it. This makes the model more robust to outliers and less prone to over-fitting
- - Each tree is also given a **randomized** subset of features it can decide on rather than given the full feature set. This reduces correlation between trees by allowing them to only make decision rules on a subset of features and reduces the over-fitting.

$O_{train}(ndth)$, $O_{test}(tmh)$
m = the number of trees (hyper-parameter), h = the depth of the trees (hyper-parameter)

Kmeans

- Unsupervised model that groups data into k clusters. Mainly used for clustering data and vector quantization. k is a hyper parameter of the model. Below is how the algorithm trains:

1. Select k random data points from data as starting cluster means.
2. Assign examples to a cluster based on smallest euclidean distance to mean.
3. Update cluster means using the labels examples, and assess change in mean.
4. If the difference is significant, go to step 2 and continue.

- The model uses random initialization so it can create sub optimal clusters, random restarts have to be performed to to find optimal clustering. Random restarts is the process of training the model several times and finding the most common cluster means calculated.
- The clusters formed by this model are always convex clusters. This means that the model does not well for data that has clsuters intertwined.

$O_{train}(ndki)$,
k = number of clusters (hyper parameter), i = number of iterations required for convergence

---

Density Based Clustering DBSCAN

- Unsupervised model that finds clusters of data by observing the neighbours it has. The model has two hyperparamters ϵ and minNeighbours:

1. Identify core points by finding points that have at least minNeighbors within the ϵ neighborhood.
2. Form clusters by connecting all core points that are neighbors to a cluster, ignoring non-core points.
3. Assign non-core points to the nearest assigned cluster if they are within the ϵ neighborhood of a assigned core point, otherwise label them as noise.

- Not affected by random intializations.
- Small epsilon with low minNeighbours: Will result in many small clusters and a lot of noise.
- Large epsilon with high minNeighbours: Might miss small, dense clusters and merge different clusters into one.
- Small epsilon with high minNeighbours: Only very dense and tightly packed regions will form clusters, with most points marked as noise.
- Large epsilon with low minNeighbours: Likely to create large, loosely defined clusters that may not be meaningful.

$O_{train}(dn^2)$

Hierarchical Clustering

- Unsupervised model that finds hierarchical clusters. Most common approach is agglomerative clustering:

1. Each point is its on cluster and define it as first layer
2. Merge the two closest clusters, define new clusters in a layer
3. Repeat step 2 until all points are in one cluster

$O_{train}(dn^3)$

Linear Regression

- A method of predicting continuous labels from data. it attempts to following equation: $y = Xw$
- For training, the objective function needs to be solved to find the weight vector w, this is done using least squares: $f(w) = \sum_{i=1}^{n}(w^T x_i - y_i)^2$
- solving this system of normal questions starts finding the gradient vector of $f(w)$ and setting it to a zero vector and finding w: $(X^T X)w = (X^T y)$
- One problem with least squares linear regression is that is suffers greatly with large outliers.

$O_{train}(nd^2 + d^3)$, $O_{test}(td)$

Nonlinear Regression

- Very similar to linear regression except X is transformed. Here the transformation could be to consider the intercept or adding polynomial basis or adding trig, log, exponential functions.
- for polynomial basis, the hyper-parameter is p, which indicates what order polynomial to transform the dataset
- Usually least squares or robust/brittle regression can be applied.
- Another transform that can be applied to X, is **Gaussian Radial Basis Functions RBFs**. Here each data point is represented with an RBF, the resulting Z matrix is of size (n x n). When the transform is used on the testing data to produce $\tilde{Z}$, the size of the matrix is (t x n). This transform is non-parametric:

$z_i = (g(||x_i - x_1||), g(||x_i - x_2||), ..., g(||x_i = x_n||))$, $g(\epsilon) = exp(-0.5\epsilon^2/\sigma^2)$
the hyper parameter is σ^2, which is the width of the "bump". Smaller width means more complicated model. RBF basis with L2 regularization and cross-validation to choose σ and λ has very good results but is computational expensive to train. When fitting regression weights w represent the heights and sign of each bump, regularization is used during fitting so λ is another hyper parameter

---

Robust Regression

- In robust regression we change out the objective function and the method used to solve for the weight vector w.
- Hyper-parameters are learning rate α and ϵ
- The objective function is changed from least squares (L2 norm) to absolute error (L1 norm) and we approximate L1 with huber loss:
$f(w) = \sum_{i=1}^{n} |w^T x_i - y_i| = \sum_{i=1}^{n} r_i$

$$r_i = \begin{cases} \frac{1}{2}(w^T x_i - y_i)^2 & \text{if } |w^T x_i - y_i| \leq \epsilon, \\ \delta\left(|w^T x_i - y_i| - \frac{1}{2}\epsilon\right) & \text{otherwise} \end{cases}$$

- Here we cannot use the normal system to solve for the weights vector w, so we have to use **gradient descent**.
- Robust regression is more robust against outliers, and is less influenced.

Brittle Regression

- Similar to above, we use a different objective function and gradient descent to find w. Here we want to use the L_∞ norm and must approximate with log-sum-exp:
- Hyper-parameters are learning rate α
$f(w) = max_{i \epsilon n} |w^t x_i - y_i| = log(\Sigma_{i=1}^{n} exp(w^t x_i - y_i))$
- this is brittle recession and it is extra sensitive to outliers.

Regularized Regression Trees

Imagine tree regression models, where first model splits data set and then each data set is analyzed by subsequent models. Regression tree is like a decision tree except it can predict a continuous value. Now ensemble them together (like random forest) and apply boosting, where one one tree predicts a value and then subsequent trees predict a residual for the first tree. If one applies L0 and L2 regularization to this model, one gets **XGBoost**

Feature Manipulation

Having too many features in your dataset and training on all of them can lead to overfitting. Need to figure out which is best features to training on. Here are some ways to manipulate features:

- **Association rules**
- **L0-regularization** apply L0-norm on the weights see regularization section
- **Apriori/forward selection algorithm**:

1. start with empty set of features, S, and calculate score.
2. find scores using each feature, j U S, initially this will only test one feature.
3. find best scoring j, and add j to S. and then repeat 2 with the other features. Keep doing this until score does not improve.

runtime of $O(d^2)$

Standardizing features is another trick we can use to improve our error. Scaling does not matter for decision trees, naive bayes, and least squares. But it does matter for KNN and regularized least squares. Here first compute the mean and standard deviation value for a given feature in our test data μ_j, σ_j. Then for replace each x_{ij} with $(x_{ij} - \mu_j)/\sigma_j$. When standardizing test data, use the μ_j, σ_j calculated from the test data. For cross validation, calculate and apply μ_j, σ_j for the folds you train on and apply to the fold you validate on.

Standardizing target is replacing y_i with $(y_i - \mu_y)/\sigma_y$. Again follow the rule of using the training μ_y and σ_y on the test data, and remember to unstandardize after you are done.

Feature Engineering:

For counting based methods (naive bayes and decision tree) using discretization or binning for features is popular for reducing overfitting.

For distance based moethods (KNN), using standardization on features is helpful for reducing overfitting.

For regression methods, non-linear transforms help for reduce overfitting.

Kernal Trick: A way to perform polynomial transformation easily on several features with out burdening calculations. This allows us to find a seperating boundry on a higher dimension. Transform X by finding $K = XX^T; k_{ij} = (1 + x_i^T x_j)^p$ where x_i, x_j are two separate examples. Here K is (n x n), cost of computing K is $O(n^2 d)$. You would train model using K, predict by calculating $\tilde{K} = \tilde{X} X^T$ and then $\hat{y} = \tilde{K}(K + \lambda I)^{-1} y$. Can also apply RBFs $K_{ij} = exp(-0.5(1 + x_i^T x_j)^2)$ for infinite dimensional basis, RBF K costs $O(n^2)$ to find.

Regularization

This is a way to prevent overfitting. Regularization adds a penalty to the loss function such that the weights impact the penalty. λ is called the regularization parameter and it must be > 0, ideally between 1 and $\sqrt{n}$, it controls the strength of regularization. In terms of fundamental trade off, regularization will increase training error but decrease the generalization gap, and can decrease test error.

L2-regularization:
$f(w) = 0.5||Xw - y||^2 + \lambda||w||^2$
$\nabla f(w) = X^T Xw - X^t y + \lambda w \rightarrow w = (X^T x + \lambda I)^- 1 X^T y$
**can also be used in GD and SGD
the idea here is to penalize slope, so take the lower slope that best fits. We do square L2 since it is differentiable at 0. Penalty insignificant at small w values.

L1-regularization:
$f(w) = 0.5||Xw - y||^2 + \lambda||w||_1$
Regularizes and also selects features (encourages sparsity). Penalty always proportional to distance from 0. Has to be solved with GD, SGD.

L0-regularization:
$f(w) = 0.5||Xw - y||^2 + \lambda||w||_0$
Encourages elements of w to be exactly zero (sparsity). Penalty always the same until w is zero. Has to be solved with GD, SGD.

Linear Classifiers

Classification with linear regression models. Very widely used, provides good test error, fast to compute, and smoother prediction compared to random forest. Here are labels are +1 or -1. Predictions will look like this $\hat{y} = sign(o_i) = sign(w^t x_i)$. We need a special 0-1 loss function here since we only want to fit the w by assessing if it made the right classification or not, not how far off the predicted value is from +1.

Perceptron Algorithm: This will find a perfect classifier if it exists.

1. $w = 0$
2. go through examples in any order until you make mistake predicting y_i. set $w^{t+1} = w^t + y_i x_i$
3. keep going until you make no mistakes.

Hinge loss, convex loss function centered around y=-1. Cannot use $-y_i w^T x_i$ as it has a degenerate solution of $f(0) = 0$, so we use $1 - y_i w^T x_i$ instead:
$f(w) = \Sigma_{i=1}^{n} max(0, 1 - y_i w^T x_i)$

Support Vector Machine is basically Hinge loss with L2 regularization. Aims to maximize the margin:
$f(w) = \Sigma_{i=1}^{n} max(0, 1 - y_i w^T x_i) + 0.5\lambda||w||^2$

Logistic loss, use log-sum-exp instead, provides a smoother 0-1 loss. Solve with GD/SGD, remember to add regularization:
$f(w) = \Sigma_{i=1}^{n} log(1 + exp(-y_i w^t x_i))$

probabilistic classifier
easy to convert to from a linear classifier. Replace $\hat{y} = sign(o_i)$ with sigmoid function $\hat{y} = 1/(1 + exp(-o_i))$

Multi class Classifier
Use the "one vs all" method where separate classifiers for each label are ensembled together, and the predicted output is the label with the highest score (do not apply sign function). Training will product a matrix W where it is (k x d), so each row is the weights for one label. If using original features, classification regions are convex but if using RBFs as features then regions are non-convex:
"sum": $f(w) = \Sigma_{i=1}^{n} \Sigma_{y_i \neq c}^{k} max(0, 1 - y_i w^T x_i + y_c w_c^T x_c)$
Penalizes for each 'c' that violates constraint. Work better than "max".
"max": $f(w) = \Sigma_{i=1}^{n} max_{y_i \neq c}(max(0, 1 - y_i w^T x_i + y_c w_c^T x_c))$
Penalizes for one 'c' that violates constraint the most. More popular choice.
If we add L2 regularization, then we get **Multi-class SVM**

multi class probabilistic classifier Use soft max function:
$p(y = c|z_1, ..., Z_c) = exp(z_c)/\Sigma_{c=1}^{k} exp(z_c)$

Other important things to remember

- Golden rule of machine learning it to never train the model with the test set
- The fundamental trade-off states that as your model gets more complex, the E_{gap} (approximation error) get bigger and E_{train} will get smaller. This equation is:
$E_{test} = E_{gap} + E_{train}$ where $E_{gap} = E_{test} - E_{test}$
- Models get more complex depending on variation of the hyper parameter and how many features the dataset has.
- IID is an assumption commonly made about datasets which enables machine learning to exist.
- Optimization bias can happen with picking a model or tuning a hyper parameter. Generally it happens because a model got lucky with a particular validation set and it seems like the best choice. The way to mitigate this is using cross-validation.
- Some useful tricks for showing a function is convex:
- - 1-variable, twice-differentiable function is convex if $0 \leq f"(w)$ for all w.
- - A convex function multiplied by non-negative constant is convex.
- - Linear functions are convex.
- - Norms and squared norms are convex, L0 norm not-convex.
- - The sum of convex functions is a convex function.
- - The max of convex functions is a convex function.
- - Composition of a convex function and a linear function is convex.
- - composition of convex function with a convex function is not always convex, E.g. if f and g are convex, $f(g(w))$ may not be convex.
- - multiplication of convex function may not be convex

PCA

PCA is a latent factor model useful for dimensionality reduction, visualization, factor discovery, outlier detection. It is comprised of two matrices that you get from the data. PCA is considered non-parametric. In PCA you define k (number of principle components), which should be $k \leq d$. After defining k you reduce the dimensions of your data set by $d - k$.

W (k x d) with each row denoted as w_c^T is part, principle components, or factors. Each column is denoted as w^d. W represents the best lower dimensional hyper plan that fits the data. This is found by minimizing square distance on all dimensions.

Z (n x k) each row denoted as z_i^T denoted part weights, factor loadings, or features. Z is the coordinates for each data point in X on the lower dimensional hyper plan W.

X approximations: $x_{ij} \approx z_i^T w^j =< w^j, z_i >$; $x_i \approx W^T z_i$; $X \approx ZW$

The Z matrix can be computed from some input data and then used in linear regression model to predict y.

PCA Objective Function:

$f(W, Z) = \Sigma_{i=1}^n \Sigma_{j=1}^d (< w^j, z_i > -x_{ij})^2 = \Sigma_{i=1}^n ||W^T z_i - x_i||^2 = ||ZW - X||_F$

*Assuming X is centered by $X - \mu$, where μ is the average across all data points in X. Here each column of X has a mean of zero.

Once training is complete the model stores W and μ. For prediction do the following:

1. Center $\tilde{X}$ with μ via $\tilde{X} - \mu$
2. Find $\tilde{Z}$ by minimizing squared error: $\tilde{Z} = \tilde{X} W^T (WW^T)^{-1}$
3. Reconstruction error is $||\tilde{Z}W - \tilde{X}||_F^2$
4. For a given k we can find the variance remaining $\frac{||ZW-X||_F^2}{||X||_F^2}$. If you select $k = d$ then it is 0 but if you select $k = 0$ then it is 1. So variance explained is just $1 - \frac{||ZW-X||_F^2}{||X||_F^2}$. Remember when calculating variance explained/remaining for $k = 2$, also includes $k = 0$ and $k = 1$

Forbinius norm of X is this: $||X||_F^2 = \Sigma_{i=1}^n \Sigma_{j=1}^d x_{ij}^2$

NON-UNIQUENESS: In PCA, the principle components are ordered by how much variance they explain. So $k = 1$ explains the most variance, then $k = 2$, $k = 3$, etc. The last principle component explains the least amount of variance.To fix this we add normalization (l2 norm of a component is 1), orthogonality (each component is orthogonal to every other component), and sequential fitting (fit the first component, then second, ...).

METHODS TO FIX: Because of PCA's properties, it has a non-unique global optimum. This is due the fact that each component can be scaled, rotated, or swapped with another components with no effect to the results. so have to come up with crafty solutions for finding W. This is enforced via singular value decomposition, gradient descent, stochastic gradient descent, or alternating minimization.

alternating minimization, require us to fix Z and find optimal W via GD or SGD, then fix W and find optimal Z via GD or SGD, this will converge to a local optimum. But with random initializations of W and Z, it will find a global optimum.

for finding W via GD: $\nabla_W f(W, Z) = Z^T ZW - Z^T X$

for finding Z via GD: $\nabla_Z f(W, Z) = ZWW^T - X^T W^T$

GD can get computationally expensive ($O(ndk + d^2 k)$ per iteration?) but SGD only costs $O(k)$ per iteration.

Variations of PCA:

- Use logistic loss in the objective function for a binary PCA model
- Use absolute loss in the objective function for robust PCA model
- Add L2 regularization to objective function, this helps improve PCA when dealing with noisey or high dimensional data by penalizing on big components or parts: $f(W, Z) = ||ZW - X||_F + 0.5\lambda_1 ||W||_F + 0.5\lambda_2 ||Z||_F$

Recommender Systems

How Netflix recommends shows to us. Two methods:

- Content based filtering, supervised learning, where features are extracted from samples of users and items and using to predict rating for new users/ratings. Can predict on new users/movies but cannot learn each user/movie. $Y = XW^T$
- Collaborative filtering, unsupervised learning, only have ratings from users of a particular item, but no features. Learns vectors for each user, but cannot predict on new users. Here we are trying to fill in unknown ratings a user has for an item based on the labels we have. $Y \approx ZW + \mu$, find Z and W using L2 regularized PCA, remeber to center Y:

$f(W, Z) = ||ZW - Y||_F + 0.5\lambda_1 ||W||_F + 0.5\lambda_2 ||Z||_F$

Multidimensional scaling and t-Distributed Stochastic Neighbor Embedding

In PCA we can visualize the lower dimensional data by plot the points in Z. Alternative to PCA there is Multidimensional scaling, MDS. One would use MDS if they wish to preserve relationships between data points that are not well captured by PCA's linear assumptions. In MDS, we directly optimize pixel locations of Z. MDS is non-parametric and has the following loss function for determining Z:

$f(Z) = \Sigma_{i=1}^n \Sigma_{j=i+1}^n (||z_i - z_j|| - ||x_i - x_j||)^2$

One problem with MDS is that it can focus on large distances sometimes and crowd together all the data points.

t-Distributed Stochastic Neighbor Embedding or t-SNE, is an amazing alternative that will preserve the local distances to neighboring data points for each data point. Preserving this information allows the resulting Z to have clusters for related data points. Some examples this makes sense is in application to MINST data set, or data sets that have a "swiss roll" structure.

Gradient Descent and Stochastic Gradient Descent

Gradient descent is an iterative method for solving w:

1. guess an initial value for w^0
2. generate the next weight using the formula: $w^t = w^{t-1} - \alpha \nabla f(w^{t-1})$

α = learning rate

3. check if progress was made by checking: $|\nabla f(w^t)| \leq \epsilon$

- gradient descent will change the time complexity to $O_{train}(ndt)$ where t is the number of iterations. Gradient descent makes sense to use if there is a large number of features

Stochastic gradient descent looks to reduce computational load by only calculating gradient on one example, i:

$w^t = w^{t-1} - \alpha^{t-1} \nabla f_i(w^{t-1})$

This works because the expectation of $\nabla f_i(w^k)$ is $\nabla f(w^k)$. SGD will began with a random descent pattern but eventually will bounce around at a boundary of radius α^t. To get convergence we need our step size, α, to decrease with each iteration. Best option is $\alpha^t = O(1/\sqrt{t})$.

A variation of SGD is **mini-batch**, where we instead compute the gradient over a small random batch of samples. This can be parallelized on GPUs easily and allows for faster convergence.

Another variation is **early stopping** where we stope the SGD before the model starts to overfit. This can be checked by calculating the validation error after each iteration in SGD, if error stops improving stope SGD.

MLE and MAP

Maximum Likelihood Estimation MLE is a method to estimate parameters of a statistical model by maximizing the likelihood function, which measures how likely the observed data is given the parameters w in $argmax_w(p(D|w)) \equiv argmin_w(-log(p(D|w)))$. For the example of least squares, if $f(w) = 0.5||Xw - y||^2$ is the objective function, we can say $y_i = w^t x_i + \epsilon_i$ where ϵ_i is sampled from a normal distribution, this makes our likelihood function $p(y_i|x_i, w) = (1/\sqrt{(}2\pi))exp(-0.5(w^t x_i - y_i)^2)$. You can change your assumption of the distribution an apply it accordingly. We can use MLE on probabilities classifier that uses sigmoid function to give logistic loss which gives justification. MLE often

Maximum A Posteriori MAP aims to get the probability of w given our data. w in $argmax_w(p(w|D))$ here $p(w|D) \propto p(D|w)p(w)$. Again can use negative log likelihood and apply regularizer by: w in $argmin_w(-\Sigma_{i=1}^n [log(p(w|D))] - log(p(w))$. Using certain distribution for our prior and likelihood yield different results. The thing with the NLL is our loss function also.

Neural Networks

- Biologically motivated, inspired by our brains.
- Basic one layer network consists of input later (size of j), hidden layer (our zs, size of k, hyper parameter), output layer (size of 1). Each node in the hidden layer has a non-linear function. $\hat{y} = v^T h(Wx_i)$ cost is $O(kd)$, v is linear combinations of activations (k x 1), h is non-linear function (often sigmoid), z_i is Wx_i, W is (k x d). Without non-linear function, our model degenerates into a linear model.
- other non-linear transforms are ReLU.
- Bias can be added by adding an extra node to the input later and setting it to one and adding an extra node before the output that has v applied to.
- For binary classification, we will need to apply sign function at the end.
- **deep learning** is done by adding extra layers of z and h in to the model. Define number of layers with m. Each layer must have a matrix of weights and a non-linear transform. Increasing depth will decrease the training error but could poorly approximate test error (lead to over fitting). Using L2 regularization in DNNs is helpful (also known as weight decay).
- Training of a model can be done through **back propagation** where we first compute the gradients with respect to v, then compute for W_m, then W_{m-1}, up until the end. **forward propagation** is when we compute a predicted value. Both of these directions have the same cost $O(dk + mk^2)$.
- SGD is utilized in training, but the problem is that the function is non-convex and can suffer from vanishing gradients. Sometimes need to do random initialization to find best model parameters to begin with. Also can come up with more elaborate step sizes to help find most optimal minimum. Early stopping is helpful.
- Some models may use skip connection where neurons connect to other neurons much more downstream from where they are.
- CNNs seek to utilize convolution transformations to help spot complex features. Convolutions apply spatial awareness to each point. In CNNs, there is several Convolution+ReLU and pooling layers. After this, the data is flatten and fed into a DNN for classification.
- Each time a convolution layer is added, the model will learn more complex features from the data. Convolution layers use a W matrix to act like several convolutions.
- Convolution+ReLU layers do not reduce the dimensions of the data (x, y) but will add layers to it (depending on number of filters). Pooling layers will reduce the dimensions of the data (x, y).
- Pooling layers combine results from convolutions, most comment technique is to use "max pooling", where we take the max value of one area and then move to the next area.

Outlier Detection

Here are ways to do outlier detection:

1. Model based - use z score to see if point makes sense or not. Hard to define outlier with some models.
2. Graphical based 0 look at graph and decide. Hard to do in higher dimensions.
3. Cluster based - use clustering models. This does not work with noisy data.
4. Distance based (KNN) - using outlier rations for the distance. Will not work with dense data,
5. supervised based - use a ML model, issue here is that what if you have a new time of outlier the model has not seen?