Loss function convexity and gradient descent:

1. In linear regression the loss function (MSE) is always a bowl shaped convex function and gradient descent can always find the global minimum (the minimum value is not always 0 but the value where the best fit can be found).

2. In logistic regression if we use the MSE as the loss function then it will not be a convex function because the hypothesis function is non-linear (sigmoid function). Thus, it will be difficult for gradient descent to find the global minimum. However, when using the cross-entropy loss (log loss) then the loss function is convex and gradient descent easily converges.

3. We should always use a convex loss optimization functions so that in our problems gradient descent can converge and have no problems to find the global optima (local optima free). MSE and log loss (cross-entropy loss) are convex functions (great for gradient descent).

4. In NNs the loss function is not convex and it's very usual to stuck in a local optima.

5. SVMs have convex loss function and usually we find a solution close to global minimum.

Advanced Optimization methods: For all derivative-based optimization algorithms (including gradient descent) we need to calculate the objective loss function $J(\theta)$ and its gradient. Also, some advanced optimization methods, e.g. conjugate gradient, BFGS, L-BFGS converge either faster or manually/automatically pick the learning rate $\boldsymbol{\alpha}$ by running internally the line search method. No need to understand the inner workings of these algorithms. Many libraries support these and can be used to find the θ parameters that minimize $J(\theta)$.

Non-linear fitting: Linear/Logistic regression can be used to fit/separate data in a non-linear way using extra polynomial terms (quadratic, cubic, etc). However, it is difficult to fit/separate very complex non-linear data. When the features are many (e.g. images in computer vision) then these polynomial terms are enormous and computationally expensive (e.g. for a 100x100 grayscale images we need 100.000 quadratic terms). So, we can use other more powerful non-linear approximators/classifiers (e.g. SVM, neural networks).

Regularization to reduce overfitting: Overfitting can be reduced in linear/logistic regression methods by either reducing the number of features (dimensionality reduction: a) projection, b) feature selection) or by adding L1/L2 regularization terms in the loss function to reduce the magnitude of the hypothesis parameters. Regularization terms smooth the fitting curve and reduce the overfitting.

Issues with features: Missing feature values, Highly correlated features, Features for the same information with different representations (foot length in cm and inch), Features with very low variance, All features with same value.

Symmetry Breaking in NNs: When the weights of a layer are initialized to the same value (e.g. 0) then all neurons of the layer produce the same output in the forward propagation. Also, when doing backpropagation the derivatives changes of the weights are also the same values. This means that all the neurons of a layer learn the same thing. Furthermore, when gradient descent ends the weight matrixes contain the same values. This problem can be solved by initializing all weights with small random values.

Model evaluation: We can evaluate a model using the training/test (70%-30%) split strategy with an evaluation metric (e.g. accuracy/misclassification rate). We first train the model with the training set and then measure the performance of the trained model for the unseen test set.

Model selection: When we want to select between various models parametrized by a hyper-parameter setting (e.g. polynomial degree level, regularization λ) we use the training/validation/test (60%-20%-20%) split strategy. We first train the models with the training set and then select the best model based on the validation error. Last, we get the generalization indication on the test set. In general we can plot the training and validation error curves w.r.t to a hyper-parameter setting that we are trying to do model selection. With such a plot we can diagnose for which setting values we have high variance, high bias or a good fit. The regularization term is used only when we train the model. When we report training/validation errors we skip it from the loss equation.

Diagnosing high variance/high bias: If our model does not perform well on test set we can try to diagnose if it is a high bias or high variance problem. When both the training and validation errors are high then we have a high-bias model (underfitting). When we have a high validation error and low training error we have a high-variance model (overfitting). When both the errors are low then we might have a good fit.

Learning curves and Bias/Variance: A technique to diagnose if a model suffers from bias, variance or both. Plot training and validation loss w.r.t to different training set sizes (for each size we provide average losses from multiple trained models with random examples). If both training and validation losses are high then we have high bias. If training loss is low and validation loss is high (big gap between them) then we have high variance.

Techniques to try / not try in case of high bias / high variance:

When we have high variance / overfitting / complex model for our data:

- Getting more data helps to fit better the data and reduce the validation error
- Keeping only informative features with dimensionality reduction can help to fit better
- Reduce model complexity, trainable parameters
- Adding additional features (e.g. polynomial terms, or other new features) usually don’t help
- Increase regularization term λ to reduce model complexity

When we have high bias / underfitting / simple model for our data:

- Getting more data cannot help to have a better fit
- Dimensionality reduction usually don’t help
- Train longer might helps
- Increase model complexity, trainable parameters
- Adding additional features (e.g. polynomial terms, or other new features) might help
- Decrease regularization term λ to increase the model complexity

How to find a good fit: A good fit is when we have low bias and low variance. In such a model both training and validation losses are low. When we train a simple model in the training data the model might overfit and providing more data cannot actually help. What we need to do is to increase the complexity of the model. However, this might lead to overfitting. Providing more data with enough information to an overfitted model can really help. So, eventually for many learning algorithms when they have some reasonably **complexity**, providing **more** data with **sufficient features** can help to create a good model with low bias and low variance.

Recommended approach for building new ML systems: Always start with a proof of concept model (built it in "24h"). Evaluate the model with an evaluation metric on the test set and have it as a baseline to compare it with any new candidate improved models. If the model does not provide good validation error, plot the learning curves (validation/training error) of the model to find if it suffers from high bias or high variance. Never do new changes/experiments only by instinct/gut-feeling in order to improve the model but make decisions based only on evidence. Try to diagnose why the model does not work well using the learning curves and the confusion matrix as it helps to investigate misclassified examples. From the hard examples useful systematic patterns can be found that the model fails to catch (e.g. in MNIST 9 can be misclassified as 4).

Skewed classes problem: Simple accuracy/misclassification rate evaluation metrics some times fail to report correctly the model's performance. For example when we have skewed classes, e.g. class A size: 97% of training examples and class B size: 3% training examples, having 99% accuracy does not mean a good model because the 1% misclassification might refer to the class B examples. For such a case we use precision, recall and f1-score metrics which are able to measure correctly the aforementioned error. Having the ground-truth vector and actual predictions vector we can calculate the precision, recall and f1-score metrics. The f1-score uses precision and recall and can be used as a single real value evaluation metric. Also, from the above vectors (ground-truth and predictions) we can create the confusion matrix (for binary classification or multi-class classification) to investigate false/true positive/negatives.

Metrics (can be used on any data set):

- Accuracy (percentage, higher the better): Percentage of correct classifications on the overall examples – (TP+TN) / Total examples

- Misclassification rate (percentage, lower the better): Percentage of wrong classifications on the overall examples – (FP+FN) / Total examples

- Recall (percentage, higher the better, per class: e.g. dogs): Percentage of examples correctly classified as dogs on the overall dog examples – TP / (TP + FN) or TN / (TN + FP)
  - Example: High recall means that for the patients that really have cancer we predicted it correctly for most of them.

- Precision (percentage, higher the better, per class: e.g. dogs): Percentage of examples correctly classified as dogs on the overall examples classified as dogs – TP / (TP + FP) or TN / (TN + FN)
  - Example: High precision means that for the patients that we predicted cancer most of them have really cancer.

- f1-score (percentage, higher the better, per class): A single real value evaluation metric that combines precision and recall results. When there is a significant difference between precision and recall (e.g. high precision, low recall and vice versa) the f1-score is low. F1-score is good only when both metrics are reasonably good. Of course they cannot be both very high cause there is a tradeoff between them. When the f1-score is higher the model predicts less false negatives and false positives. We usually do model selection between various models using the f1-score.

It is possible to do model selection by changing the prediction threshold:

a) Big threshold (>= 0.90): We are confident when an example is predicted as positive
   e.g. Be confident when we predict that someone has cancer

We have more false negatives (predict that someone has no cancer but in reality has)
Precision increases (from all cancer predictions most of them are correct)
Recall decreases (from all really cancer examples we predicted correctly only some)

a) Small threshold (>= 0.30): We are confident when an example is predicted as negative
e.g. Be confident when we predict that someone has no cancer
We have more false positives (predict that someone has cancer but in reality hasn't)
Precision decreases (from all cancer predictions most of them are incorrect)
Recall increases (from all really cancer examples we predicted most of them)

Tradeoff between Precision / Recall:

- We can increase the confident that our predictions are correct but only on a small fraction of examples (high precision – low recall)

- We can increase the number of correct predictions but we cannot be enough confident that are correct (low precision – high recall)

Support Vector Machines:

liblinear executable for linear SVM, libsvm executable for non-linear SVM (kernel-based SVM)

The hypothesis function of a Support Vector Machine (SVM) predicts either 0 or 1 (discrete value) and doesn't calculate the probability (e.g. in logistic regression) of x being in the class 1, $p(y=1,x;\theta)$ which is a real value. An example x is predicted as 1 when $\theta^T x >= 1$ and as 0 when $\theta^T x <= -1$. The SVM uses a separation hyperplane with a large margin because it doesn't want to just barely classify correctly an example (e.g. y=1 and $\theta^T x = 0.001$ or y=0 and $\theta^T x = -0.001$). We want the closest data points to the decision boundary (support vectors) to be as far as possible from it. This extra large margin makes the model more robust to test/unseen data.

The loss objective function that enables that is:

$$\min_{\theta} C \sum_{i=1}^{m} \left[y^{(i)} cost_1(\theta^T x^{(i)}) + (1 - y^{(i)}) cost_0(\theta^T x^{(i)}) \right] + \frac{1}{2} \sum_{i=1}^{n} \theta_j^2$$

(cost1 and cost0 are actually Hinge losses)

This is a multi-objective function with two terms, first (A) the loss term for the training data set and second (B) for the norm of the model parameters. By minimizing B we actually maximizing the margin boundary of the separating hyperplane. Also, by minimizing A we actually separate the training data better. The 'c' parameter is a weight coefficient that controls the penalty for misclassified training examples. When 'c' is large we pressure the minimization mostly to separate the two classes. So, the margin boundary might not be so large. When 'c' is small then we pressure the minimization to find a large margin boundary but the separation of the two classes might not be so good (when the classes are not linearly separable or have some outliers this is good to happen). When the margin boundary is small then we separate good our training data but our model is not consistent to test data. However, when the margin boundary is large we might not have so good separation on training data but the model is more consistent to the test data. We need to do model selection with the 'c' parameter and training/validation/test data sets in order to find the best model.

Kernel-based SVM:

We already know that by adding extra polynomial terms (e.g. in linear/logistic regression) we can create complex non-linear functions that fit/separate non-linear data. However, when the dimensions of the input feature space are too many (e.g. images in computer vision) then it is very difficult to compute all the polynomial features for the feature values. To overcome this problem in the SVMs we use kernel functions.

Kernels are used in order to represent complex non-linear decision boundaries for separating non-linear data. Using a kernel function we increase the dimensionality of the feature vectors (even to infinite dimensions) hoping that our new data will be linearly separated with a hyperplane. A kernel k(x, l) is a similarity function between a point x and another one l which is is called 'landmark'. For any $x^{(i)}$ data point we can create a new feature vector with n features $f^{(i)} = [f^{(i)}_1, f^{(i)}_2, \dots f^{(i)}_n]^T$ using n landmarks $l^{(1)}, l^{(2)}, \dots, l^{(n)}$ as $f^{(i)}_1=k(x^{(i)}, l^{(1)})$, $f^{(i)}_2=k(x^{(i)}, l^{(2)})$, …, $f^{(i)}_n=k(x^{(i)}, l^{(n)})$ where k(x, l) is a kernel function (e.g. Gaussian RBF, Sigmoid, Linear [same as using no kernel], Polynomial).

Most of the times we need to use many landmarks cause we want to increase very much the dimensionality of our data. So, how do we select the $l^{(i)}$ landmarks? One thing we can do is to use all the training data points as landmarks.

So, if the training size is m examples then for each training $x^{(i)}$ example we will create a transformed feature vector as $f^{(i)} = [f^{(i)}_1, f^{(i)}_2, \dots f^{(i)}_m]^T$ where $f^{(i)}_j=k(x^{(i)}, l^{(j)})$. Also, we should note that $f^{(i)}_i$ will always be 1 cause $x^{(i)}=l^{(i)}$. The new features of $f^{(i)}$ now measure how close (using some similarity kernel, e.g. Gaussian RBF) the training example $x^{(i)}$ is to all the training data points (including itself $f^{(i)}_i=1$). When the trained model is saved we actually save also the landmarks, the kernel and its hyperparameters that will transform any test example in the same way training data were transformed.

The objective function of kernel-based SVM:

$$\min_{\theta} C \sum_{i=1}^{m} y^{(i)} cost_1(\theta^T f^{(i)}) + (1 - y^{(i)}) cost_0(\theta^T f^{(i)}) + \frac{1}{2} \sum_{j=1}^{n} \theta_j^2$$

Also, in the kernel-based SVM we usually do grid search with the 'c' hyper-parameter and any kernel parameters (e.g. σ^2 for Gaussian RBF kernel) to find the configuration that gives the best results (as variance/bias tradeoff is concerned). When 'c' is large then this leads to high variance / low bias and when is small leads to low variance / high bias. Also when σ^2 is small then this leads to high variance / low bias (the decision boundary encloses tightly the training data the class) and when is large leads to low variance / high bias (the decision boundary does not enclose tightly the training data the class).

Tips:

1. When we have a **small** (m=10...1000) size of training examples and **many** (n=10000) features it is not good to use a kernel to increase the dimensionality even more because this could probably lead to a very complex model model that will probably overfit (high variance / low bias). In such a case it might be more reasonable to use logistic regression or SVM without a kernel (linear kernel) and not try to fix a complex non-linear function.

2. When we have an **intermediate** (m=10...10000) size of training examples and **few** (n=1...1000) features we can use SVM with a Gaussian kernel to increase the dimensionality hoping that in the new high dimensional feature space the data could be linearly separable with a hyperplane.

3. When we have an **enormous** (m=50000+) size of training examples and **few** (n=1.1000) features it is very computationally expensive to use SVM with a kernel to increase the dimensionality cause there will be so many landmarks and features calculate. In this case we could try to add manually new features and then perform logistic regression or SVM without a kernel.

4. When using the Gaussian kernel we should always normalize the features first. This is important because in the Gaussian kernel we calculate and use the distance between a data point x and a landmark, k(x, l). If the two data points differ significant in a feature with small scale then this difference will not be significant if there are also features with higher scale (e.g. a=[10000, 1], b=[10010, 3]).

5. Polynomial kernel usually is used when our data points are strictly non-negative.

6. The most well-known and used kernels are Gaussian and linear.

K-Means clustering:

What happens if a cluster ends with no items assigned while the algorithm is running? We can either eliminate the cluster (so we end up with K-1 clusters) or we can randomly re-initialized a data point as the new cluster centroid. K-means can be used for separated classes as well as non-separated classes. With non-separated classes the algorithm will actually converge to a clustering but it will not be always the most intelligent one. So, we have to diagnose/evaluate the quality of the clustering somehow.

K-means distortion objective function:

$$J(c^{(1)},\dots,c^{(m)},\mu_1,\dots,\mu_K) = \frac{1}{m}\sum_{i=1}^{m}||x^{(i)} - \mu_{c^{(i)}}||^2$$

The steps of K-means algorithm: First randomly initialize the cluster centroids and then iteratively perform a) cluster assignment step, and b) move centroid step. The cluster assignment step actually minimizes the J distortion function w.r.t. $c^{(j)}$ parameters (data point clusterings) holding the μ_i parameters (cluster centroids) as constants and the move centroid step actually minimizes the J distortion function w.r.t. μ_i parameters holding the $c^{(j)}$ parameters as constants.

Initialization of algorithm / Problems with local optima: The initial step of the algorithms is the random initialization of K cluster centroids. One of the most well-known solutions is to select randomly K distinct data points from the data set and use these as the initial K cluster centroids. However, with different initializations the algorithm can lead to different clustering results (e.g. some initializations may stuck in local optima while other might be better).

How do we know if a clustering is working? The distortion error has to decrease as the number of iterations increases.

Model selection with K-means algorithm:

a) One thing we can do to increase the possibility of finding a good solution is to automatically try multiple times the algorithm with different random initializations and at the end select the one that minimizes the distortion function most. When we want a clustering with **few clusters** (2...10) trying multiple times the K-means algorithm with different initializations can really help to find a better

solution. However, when we want **many clusters** (e.g. 50+) multiple runs of the algorithm give more or less the same solution.

How to choose the K number of clusters: It is done manually and needs human insight. One way is to use the "elbow method" in which we plot for different k values the distortion error (should be the best value from multiple random initializations) and select as a good candidate the k value that gives the "elbow" in the error curve. However, sometimes there isn't a clear "elbow" point in the curve. So, another way is to use a different evaluation metric instead of distortion error to measure the quality of our clustering. We usually use an evaluation metric that serves the purpose of clustering (e.g. when clustering an image in order to compress it we can use a compression evaluation metric to select the number of clusters that give the best results).

K-means example for market segmentation: Suppose that we want to design t-shirts for a targeted population and we want the t-shirt sizes to be suitable for the most people. For each individual we have the height and weight features. We are going to make t-shirts of 3 sizes (small, medium, high) and we want to group the individuals of the population to 3 predefined clusters hoping that the algorithm will catch reasonably any small, medium high sub populations. After evaluating the quality of the clustering we might be able to know the ratio of small, medium, high t-shirts (from the cluster sizes, e.g. 50% medium, 25% large, 25% small) that we are going to make and also we can find the dimensions of the t-shirts so that can suit the most individuals (using the height and weight features of the specific cluster).

Dimensionality reduction with PCA

Reasons for dimensionality reduction: data compression to either reduce memory/disk usage or speedup learning algorithms cause less features are used, visualization in 2D/3D for data understanding, have some data representation in low-dimensional space.

PCA:

The algorithm finds a low-dimensional surface/subspace (line, plane, hyperplane) in which the initial data points are projected and the sum of squared errors / projection error / MSE between the initial data points (from the input feature space) and the projected data points (from the low-dimensional feature space) is minimized. The same algorithm can approximately reconstruct the initial data points.

If we have n-dimensional data points and we want to project them to a k-dimensional space (where k<n) we need first to normalize the features so that they have 0 mean (mean normalization) and same scale (divide by max-min range or standard deviation). After that, we need to find the k eigenvectors/principal components (represented in the n-dimensional space) that give the most variance (eigenvalues) of data or minimize the projection error.

To calculate the eigenvectors/principle components we need to do eigenanalysis on the covariance NxN matrix of the input data (code: [eigenVectors, eigenValues, _] = svd(sigma) or [eigenVectors, eigenValues] = eig(sigma) where sigma is the covariance matrix: $1/m * X^T * X$ (mean is **0**) where X is the data matrix). Both eigenVectors and eigenValues are NxN matrixes.

Projection of a data point x: $z = U_{reduced}' * x$, where $U_{reduced}$ = eigenVectors(:, 1:k)

Reconstruction of a data point x from z: $x\_approximate = U_{reduced} * z$

How to choose k (no. of eigenvectors/principle components)?

a) By using a scree plot we choose the smallest k in order to keep the most of data information/variance. More specifically, the scree plot is a curve with the variances (eigenvalues) and principle components and we can use it to see how many k principle components we will retain for our needs.

b) Automatically, we can find the appropriate k value (no. principle components) that retain a specific percentage of information/variance by using the eigenValues from [eigenVectors, eigenValues, _] = svd(sigma) which is actually a diagonal NxN matrix containing the variance for each eigenVector in the diagonal elements. We can choose the smallest k (try incrementally various k values, e.g. 1, 2, 3, etc) so that the ratio of [total of k variances / total of n variances] >= a percentage of retained information/variance (e.g. retain 95% of information/variance).

Notes: When using PCA in a supervised learning problem we perform PCA on the training set and we use the project z data points in place of old x data points. When we evaluate on test or validation sets we transform the data using the same eigenvectors from PCA on training set. Also, overfitting can be reduced by performing dimensionality reduction using PCA. However, it is more appropriate to do regularization to reduce overfitting.

Anomaly Detection

Usages: Can be used either to remove outliers (for helping learning algorithms to be more accurate), investigate unusual outliers (to find unusual behaviors) or detect outliers (fraud detection).

How it works: The anomaly detection algorithm builds a model that outputs p(x) which is the probability that an example x is normal. When p(x) is smaller than a threshold ε we assume that x as an outlier.

Model p(x) assuming Gaussian distribution of features (from now on this will be called as "original model"): The anomaly detection algorithm can model $p(x) = p(x1;\mu1,\sigma1)*...*p(xn;\mu n,\sigma n)$ in which we calculate the Gaussian distributions (by estimating first the appropriate mean and σ parameters) of all independent features of data. We assume that each feature is distributed according to a Gaussian distribution. The p(x) (probability of x being a normal example) is going to be small when one ore more $p(xi;\mu i,\sigma i)$ are small (this happens when a feature value is an outlier).

Evaluation of the model when we have labeled data: The algorithm can be used both with labeled and unlabeled data either as an unsupervised or a supervised method. When we have labeled data we can evaluate the anomaly detection model with an evaluation metric such as f1-score. The model is fitted only with the normal examples (training set should not contain outliers) and only test and validation sets should contain anomalous examples. Usually, the classes of normal/anomalous examples are skewed so it better to use the f1-score evaluation metric instead of accuracy. Also, by changing the ε threshold we can have either more false negatives or false positives. For that reason we can perform model selection on the hyperparameter ε on a validation set to find a good model.

When to use an anomaly detection algorithm instead of a supervised learning algorithm? When we have many examples for both normal and anomalous classes (spam detection) we can use supervised learning since the algorithm can really learn the structure of a normal or an anomalous example. However, when we have a very small number of anomalous examples (the classes are skewed) it is better to use anomaly detection and build the model using the normal examples. Also, when the anomalous/outlier examples seem that share a common structure then it would be better to use a supervised learning algorithm which will learn the concept of an anomalous example.

However, when the various outliers are dissimilar then using anomaly detection seems a better way to solve the problem.

Selecting the features to use

a) The kind of features we use for anomalous detection are very important (we tend to use features that can indicate anomalous behavior, e.g. features that can take large or small values).

b) Having Gaussian features (this is not mandatory) can help the anomaly detection algorithm. Usually, we can transform our non-Gaussian features to more Gaussian-like with transformations like log(x1), log(x2+c), sqrt(x3), x4^1/3.

c) In case we face a new anomalous example (fault aircraft engine) that our model classifies as normal (false negative) we can analyze it as a hard example in order to come up with new features that can help our model detect it as an outlier.

Model p(x) with multivariate Gaussian Distribution: When the features of data are correlated then this method models p(x) better than the original method, using an elliptical Gaussian distribution which can be not axis-aligned. This method fits the Gaussian probability distribution better to the manifold of the data (fits better when features are positive or negative correlated). The method uses the means and the variance-covariance matrix of data. The original model can be proved that is actually a specialization of multivariate Gaussian distribution model using 0 covariances in the variance-covariance matrix.

Original vs multivariate Gaussian models

a) The original model is computational cheaper vs the multivariate Gaussian model and can be used for large number of features. For the multivariate Gaussian model we need to compute the inverse of covariance matrix (which is nxn matrix) and when the number of features n is large this is a problem.

b) When we want to catch or model automatically the correlations in the data features we might need to use multivariate Gaussian model instead of the original model. However, in case we want to create manually new features that describe the correlation of data features then we can use the original model.

Recommendation Systems

Given some previous item ratings made by users try to predict item ratings that users might do for unrated items.

Content-based recommendation

- **Without ratings support**

  Assuming that we have features for the items we can recommend to a user similar items based on the features of the item in interest.

- **With ratings support**

  Assuming that we have features for the items and some ratings we can fit a separate linear regression model for each user that predicts item ratings based on previous

observations (x = item features, y = ratings). For each linear regression model the training data are related only to a specific user.

Collaborating filtering (Low-rank matrix factorization)

- When we have only the x features (e.g. how much action or romance is a movie) for the items we can learn the θ parameters (e.g. how much a user likes action or romance movies) for each user by solving an optimization problem to minimize the squared error of user ratings and predicted ratings.

- When we have only the θ parameters for the users we can learn the x features for each item by solving an optimization problem to minimize the squared error of user ratings and predicted ratings.

- When we neither have θ user parameters or x item features we can solve an optimization problem in order to find both θ and x for each user and item in order to minimize the squared error of user ratings and predicted ratings. This method automatically learns the features that minimize the objective function. The learned x item features can be used for retrieval of similar items and the learned θ parameters are useful for connecting users with similar interests.

Small variation of the algorithm for suggesting non-zero ratings to users with no ratings: Mean normalization of the ratings.

Large Scale Machine Learning (ML with massive datasets)

Using more training data always helps? In general, when we have a high bias model we cannot improve it by training it with more data. Only low bias (high variance) models can be improved using more training data. High bias models have small capacity and cannot be improved with more data (cannot learn more structure) while in low bias models the opposite occurs. Having massive amounts of training data is more important than the kind of low bias model we use. It seems that most low bias models in case of lots training data will actually converge to a good solution. However, when training a model with a very large number of examples it is computational expensive to calculate the grad of loss w.r.t. to a single parameter (cause all m examples are used for the calculation). This is the behavior of batch gradient descent (BGD). **How we can solve this problem?** We can use stochastic or mini-batch gradient descent (SGD, MBGD) and update the parameters of a model in multiple iterations using 1 example or batches of examples respectively. Actually, it is wiser to use a small number of examples first to see how it behaves. It is a good practice to train our model with e.g. 3000 examples (assume that total m is 300M) and plot the learning curve to investigate the model. If the model suffers from high bias it means that we have reached the capacity of the model and increasing the number of examples is not going to help. The next thing we need to do is to change the model to have a low bias (make the model more complex). When we have a model with low bias bringing more data examples can really help.

How do we know that SGD converges? In order to check if GD in general converges we usually plot the training error w.r.t to training iterations. In BGD we plot in each iteration the total training error (total error from all m examples). However, in SGD we usually plot an averaged error for each k iterations (we calculate the errors of the last k training examples and we average them). Also, while the training error in BGD is decreasing on each iteration, in SGD we might see some oscillations because we minimize only the error of a single example. Despite the oscillations the training error decreases in general over time. In case we increase k more the training error curve is going to be smoother.

Searching the global minimum: BGD can find the global minimum (e.g. in linear/logistic regression where the loss function is convex). However, while SGD/MBGD are faster and need less memory usually they oscillate in a solution around the global minimum (which in most of the times is acceptable). In case we want to find global minimum we can use a smaller learning rate around the global minimum. So, we can dynamically slowly decrease learning rate over time so that when we reach around the global minimum the learning rate will be very small and we will take small steps.

Online learning and map-reduce technique

Online Learning: Continuously fit a model using a continuous stream of data. There is no fixed m training set. We repeatedly use new examples one-by-one (same as stochastic gradient descent) and update the parameters of the model. We look only once a new example, update the model parameters for it then go to the next one. This method is able to catch any new trends in the data since dynamically updates the model when new examples arrived. This method is used in websites where we try to predict the behavior of users. For example, in a shipping service we can try to predict if the user is going to y=accept or y=not-accept the offered price based on the some x features (price might be included) related to the user and its shipping. If we can have access to many data examples we can model a binary classifier that predicts if a user is going to accept or not an offer the system proposes. So, in case we estimate that the user will ignore the offer we can make decisions, e.g. propose a cheaper offer.

Map-reduce: We can scale various learning algorithms (e.g. linear regression, logistic regression, NNs) that use massive data and batch gradient descent using the map-reduce technique. Both the batch gradient descent update rule and the loss function (which is mandatory in case of advanced optimization techniques) use a summation over all the examples. Whenever we have a summation that uses a massive number of examples we can parallelize it. For example, in the case of batch gradient descent, we can partition the training data into different partitions of examples and send each to a different machine or CPU/GPU core to calculate a partial summation of derivatives (this is the mapping operation). After the calculation of the partial summations we can aggregate in a single machine the results and execute the batch gradient descent update rule (this is the reduce operation).

Photo OCR

Goal: Extract/Read textual information from a photo.

Can be used for: helping blind people, autonomous car systems, automatically search massive number of photos by text, etc.

Pipeline:

1. Detect text regions/rectangles

   Sliding window technique is used with a text recognizer trained on fixed size grayscale images with enough text variance (translation, rotation, zoom, scale, etc). At the end a feature map is generated where we know the probability of each rectangle to contain text. We use an expansion operator to unite blobs of high probabilities that are close to each other. Then we place a rectangle around continuous blobs of high probabilities.

2. Character segmentation

Sliding window technique is used with a text recognizer trained on fixed size grayscale images that contain patterns of consecutive characters. Whenever we detect such a pattern we separate the characters (in the middle of rectangle).

3. Character recognition

   A generalized character classifier is used which has been trained with lots of training examples. The model is trained with examples that have enough variance (we can generate new examples by either distorting the existing ones with geometrical transformations or synthesizing new ones).

4. Optional Spell-check (in case some characters are recognized incorrectly).

Saving Time!!! With ceiling analysis! Find the component in a machine learning pipeline to invest more to have a big performance difference.