**Machine Learning Crash Course Notes**

**Week 1**

* In **unsupervised learning**, the goal is to identify meaningful patterns in the data. To accomplish this, the machine must learn from an unlabeled data set. In other words, the model has no hints how to categorize each piece of data and must infer its own rules for doing so.
* For example,
  + Here, we have two clusters. (Note that the number of clusters is arbitrary). What do these clusters represent? It can be difficult to say. Sometimes the model finds patterns in the data that you don't want it to learn, such as stereotypes or [**bias**](https://developers.google.com/machine-learning/glossary#bias_ethics).
* **Note: While it is very common, clustering is not the only type of unsupervised learning.**
* In **Reinforcement Learning** you don't collect examples with labels. Imagine you want to teach a machine to play a very basic video game and never lose. You set up the model (often called an agent in RL) with the game, and you tell the model not to get a "game over" screen. During training, the agent receives a reward when it performs this task, which is called a reward function.
* The lack of a data requirement makes RL a tempting approach. However, designing a good reward function is difficult, and **RL models are less stable and predictable** than supervised approaches.

A screenshot of a cell phone

Description automatically generated

* Many machine learning systems produce models that encode knowledge and intelligence by interpreting signals differently than humans do. A neural network might interpret a word via an **embedding**, so "tree" is understood as something like, [0.37, 0.24, 0.2] and "car" as [0.1, 0.78, 0.9]. The neural network might use these representations to do accurate translations or sentiment analysis, but a human looking at the embeddings would find them very hard to understand.
* **Hard ML Problems and Why is Hard?**
  + Clustering: How to cluster if you have 1000-dim data and you have no idea about the shape of data.
  + Anomaly Detection: How to decide a sample of data is anomaly?
  + Causation: ML can identify correlations—mutual relationships or connections between two or more things. Determining causation (one event or factor causing another) is much harder. In other words, it is easy to see that something happened, but much harder to understand why it happened. In general, you need to intervene in the world—run an experiment—to determine causation; you can't see it in purely observational data.
* **Supervised ML:** ML systems learn how to combine input to produce useful predictions on never-before-seen data.
* **Training** means creating the model. That is, you show the model labeled examples and enable the model to gradually learn the relationships between features and label.
* A **Model** defines the relationship between features and label.
* **Inference** means applying the trained model to unlabeled examples. **Inference = make predictions.**
* A **regression**model predicts continuous values.
* A **classification** model predicts discrete values.
* **Linear regression** is a method for finding the straight line or hyperplane that best fits a set of points.
* **L2 Loss (Squared Loss)** = (observation - prediction)2

A picture containing knife

Description automatically generated

A picture containing object, clock

Description automatically generated

* In general, **Linear Regression** uses **L2 Loss.**
* **NOTE: In practice, finding a "perfect" (or near-perfect) learning rate is not essential for successful model training. The goal is to find a learning rate large enough that gradient descent converges efficiently, but not so large that it never converges.**
* **Convex Problems:** They’re shaped like a giant bowl. Convex problems have only one minimum; that is, only one place where the slope is exactly 0. That minimum is where the loss function converges. Local minima = global minima. You can initiate the weights anywhere; it does converge somehow. However, many ML problems are **non-convex.** For example, neural networks. For each local minima might not be equal the global minima. Strong dependency on initial weight values.

A close up of a device

Description automatically generated A screenshot of a cell phone

Description automatically generated

* Calculating the loss function for every conceivable value of  over the entire data set would be an inefficient way of finding the convergence point. Let's examine a better mechanism—very popular in machine learning—called **gradient descent**.
* The **gradient** always points in the direction of steepest increase in the **loss** function.
* In **gradient descent**, a **batch** is the total number of examples you use to calculate the gradient in a single iteration.
* **Stochastic Gradient Descent / Online Gradient Descent:** Compute loss & gradients over one sample at a time.
* **Batch Gradient Descent:** Compute loss & gradients over entire training data set. (Loss & gradients are averaged over size of the data set.)
* **Mini-batch Gradient Descent:** Compute loss & gradients over more than 1 and less than total number of samples in the training set. (Loss & gradients are averaged over the mini-batch.)
* **Why to use SGD or Mini-Batch GD instead of BGD?**
* A very large batch may cause even a single iteration to take a very long time to compute. (Real world data sets often contain billions or hundreds of billion samples and huge number of features for each sample.)
  + Even if you choose samples from the data set randomly, a large data set with randomly sampled examples probably contains redundant data. In fact, redundancy becomes more likely as the batch size grows. Some redundancy can be useful to smooth out noisy gradients, but enormous batches tend not to carry much more predictive value than large batches.
* **Why to use Mini-Batch GD instead of SGD?**
  + Given enough iterations, SGD works but is very noisy.
* Training loss should steadily decrease, steeply at first, and then more slowly. Eventually, training loss should eventually stay steady (zero slope or nearly zero slope), which indicates that training has converged.

**A screenshot of a cell phone

Description automatically generated**

* **Lowering the learning rate** while **increasing the number of epochs or the batch size** is often a good combination.
* Setting the batch size to a very **small batch number** can also cause **instability.** First, try large batch size values. Then, decrease the batch size until you see degradation.
* **Remember:** the ideal combination of hyperparameters is data dependent, so you must always experiment and verify.
* **Quantile bucketing:** A screenshot of a cell phone

  Description automatically generated
* An **overfit** model gets a low loss during training but does a poor job predicting new data. Overfitting is caused by making a model more complex than necessary.
* **generalization bounds:**
  + the complexity of the model
  + the model's performance on training data
* The following three basic assumptions guide **generalization:**
* We draw examples **independently and identically** (**i.i.d**) at random from the distribution. In other words, examples don't influence each other. (An alternate explanation: i.i.d. is a way of referring to the randomness of variables.)
* The distribution is **stationary**; that is the distribution doesn't change within the data set.
* We draw examples from partitions from the **same distribution.**
* **To prevent Overfitting:**
* Apply Cross-Validation
* Training with more data
* Remove some features
* Stop training early
* Apply regularization
* Apply ensemble methods.

**Week 2**

* Mapping Raw Data to Features: **One Hot Encoding:**

Translate string value into a feature vector. For example, if we have a one hot encoding for street names, we’ll have a unique coefficient for each possible street. One hot encoding is extremely handy for sparse, categorical data.

**A picture containing screenshot

Description automatically generated**

* The length of this vector is equal to the number of elements in the vocabulary. This representation is called a **one-hot encoding** when a single value is 1, and a **multi-hot encoding** when multiple values are 1.
* **Avoid rarely used discrete feature values**

Good feature values should appear more than 5 or so times in a data set. Doing so enables a model to learn how this feature value relates to the label. That is, having many examples with the same discrete value gives the model a chance to see the feature in different settings, and in turn, determine when it's a good predictor for the label.

A screenshot of a social media post

Description automatically generated

* **Data Cleaning & Standardization Techniques:**
* **Scale feature values:** Scaling means converting floating-point feature values from their natural range (for example, 100 to 900) into a standard range (for example, 0 to 1 or -1 to +1). If data set has only one feature, no need to scale. Advantages of the scaling:
  + - * + Helps gradient descent converge more quickly.
        + Helps avoid the "NaN trap," in which one number in the model becomes a [NaN](https://wikipedia.org/wiki/NaN) (e.g., when a value exceeds the floating-point precision limit during training), and—due to math operations—.
        + Helps the model learn appropriate weights properly. Without feature scaling, the model will pay too much attention to the features having a wider range.
* You don't have to give every floating-point feature exactly the same scale. **Nothing terrible** will happen if Feature A is **scaled from -1 to +1** while Feature B is **scaled from -3 to +3.** However, your model will react poorly if Feature B is scaled from 5000 to 100000.
* **Minimize the influence of the extreme outliers.** One way: take the log of the values. Another way: Clipping feature values, so equal extreme values to a high/low fixed standard value.
* **Binning:** Transform continuous values into discrete. You can divide into equal regions or quantiles.

A close up of a map

Description automatically generated

* **Scrubbing:** Remove bad values from the dataset. (Remove omitted values, duplicate values, bad labels and bad feature values.)
* A **feature cross** is a synthetic feature that encodes nonlinearity in the feature space by multiplying two or more input features together. (The term cross comes from [cross product](https://wikipedia.org/wiki/Cross_product).) Let's create a feature cross named  by crossing  and .

We treat this newly minted feature cross just like any other feature. The linear formula becomes:

A linear algorithm can learn a weight for  just as it would for    and . In other words, although   encodes nonlinear information, you don’t need to change how the linear model trains to determine the value of  .

**Note:** You can create a feature cross formed by squaring a single feature.

* **Cross Product:**

**A close up of text on a white background

Description automatically generated**

* **Regularization** is what we do to **avoid overfitting** and there are a lot of different regularization strategies:
* **Early stopping:** Just stop the training before you really converge on the training data.
* **Penalize the model complicity** while we’re training.
  + **How do we define model complexity?**

Two common (and somewhat related) ways to think of model complexity:

1. Model complexity as a function of the weights of all the features in the model.
2. Model complexity as a function of the total number of features with nonzero weights. (A later module covers this approach.)

One solution to decrease complexity is that prefer smaller weights. In this regularization strategy we penalize the sum of the squared values of the weights.

**Complexity(model) = sum of the squares of the weights**

* + **L2 Regularization (Ridge Regularization / L2 Weight Decay):**

We can quantify complexity using the *L2* **regularization** formula, which defines the regularization term as the sum of the squares of all the feature weights:

A screenshot of a social media post

Description automatically generated

When we use L2 regularization it does not pay attention to the training data, but it tries to make sure that, we don’t end up with any weights that are sort of bigger than they need to be. Our **weights should be sort of centered around 0 and not too big,** normally distributed.

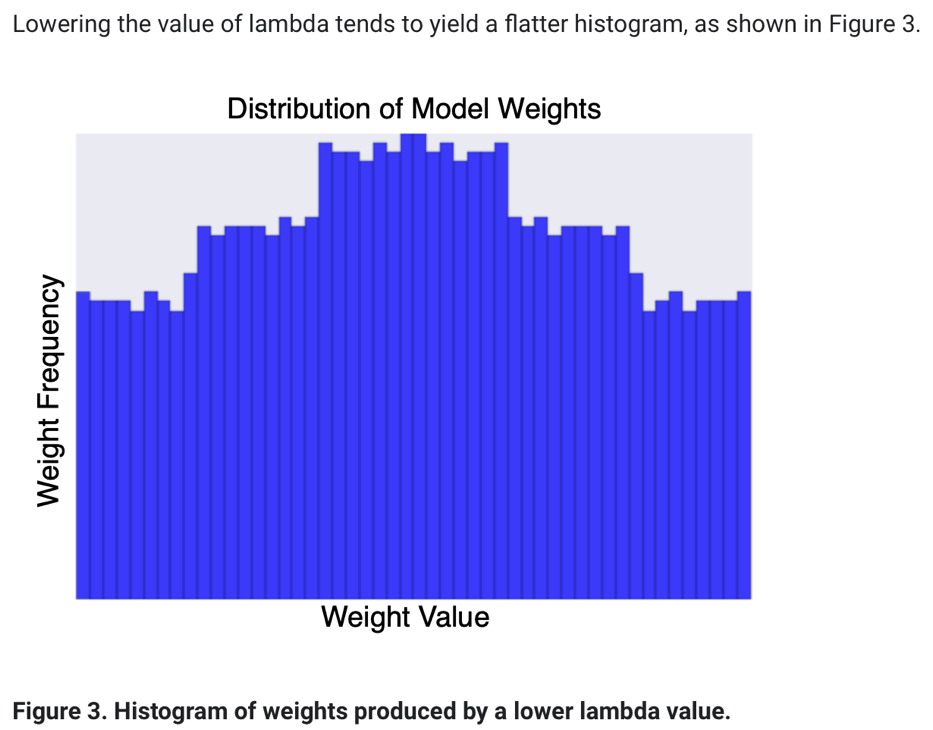
A screenshot of a cell phone

Description automatically generated

* + **How to choose L2 Regularization coefficient of Lambda ()?**

If you have lots of training data and your training data and your test data looks the same (IID), then you probably don’t need much regularization, maybe or maybe none at all. But if you don’t have much training data or if your training data and test data are kind of different, then you may want a lot of regularization. And you may want to tune that with cross validation or with a separate test set.

* + **Effects of L2 on a model:**
    - Encourages weight values toward 0 (but not exactly 0)
    - Encourages the mean of the weights toward 0, with a normal (bell-shaped or Gaussian) distribution.



* When choosing a lambda value, the goal is to strike the right balance between simplicity and training-data fit:
* If your lambda value is too high, your model will be simple, but you run the risk of *underfitting* your data. Your model won't learn enough about the training data to make useful predictions.
* If your lambda value is too low, your model will be more complex, and you run the risk of *overfitting* your data. Your model will learn too much about the particularities of the training data and won't be able to generalize to new data.
* **Logistic Regression**: Instead of predicting exactly 0 or 1, logistic regression generates a probability—a value between 0 and 1, exclusive.

**Week 3**

* **How do we evaluate classification models?**
* **Accuracy:**

**Accuracy** is a sort of metric, but for class-imbalanced problems where there is a significant disparity problem between number of positive and negative labels, useful technique is to separate out different kinds of errors and so we can combine these errors into a couple of different metrics: **Confusion Matrix:**

|  |  |  |
| --- | --- | --- |
| True Positive | The shepherd said, “Wolf!! ” | There is actually a wolf. |
| False Positive | The shepherd said, “Wolf!! ” | There is not any wolf. |
| True Negative | The shepherd did not say. | There is not any wolf. |
| False Negative | The shepherd did not say. | There is actually a wolf. |

* **Precision** (Kesinlik / Hassasiyet / Duyarlılık): **What proportion of the positive estimations are correct?** When the shepherd said, “Wolf!!” How many of times was he right? How precisely was he able to say “Wolf!!”?

* **Recall** (Geri Çağırma): **What proportion of actual positives was identified correctly?** When the wolves tried to come to the village how many did the shepherd said?
* To fully evaluate the effectiveness of a model, you just examine both precision and recall, but it’s a tough thing. It is just because, improving precision typically reduces recall and vice versa.

|  |
| --- |
| A picture containing screenshot  Description automatically generated  A close up of a clock  Description automatically generated |
| A picture containing screenshot  Description automatically generated  A close up of a clock  Description automatically generated |

* **To solve Precision vs. Recall Race 🡪 Use F1 Score:** It considers both the precision p and the recall r of the test to compute the score:

A close up of a logo

Description automatically generated

* **Classification Threshold (Decision Threshold):** A logistic regression model that returns 0.9995 for a particular email message is predicting that it is very likely to be spam. Conversely, another email message with a prediction score of 0.0003 on that same logistic regression model is very likely not spam. However, what about an email message with a prediction score of 0.6? In order to map a logistic regression value to a binary category, you must define a **classification threshold** (also called the decision threshold). A value above that threshold indicates "spam"; a value below indicates "not spam." It is tempting to assume that the classification threshold should always be 0.5, but thresholds are problem-dependent, and are therefore values that you must tune.
* **ROC Curve (Receiver Operating Characteristic Curve)** is a graph showing the performance of a classification model at all classification thresholds. This curve plots two parameters:
* **True Positive Rate**
* **False Positive Rate**

A picture containing bird, tree, flower

Description automatically generated

* **Evaluation Metrics: AUC (Area Under the ROC Curve):** AUC stands for "Area under the ROC Curve." That is, AUC measures the entire two-dimensional area underneath the entire ROC curve (think integral calculus) from (0,0) to (1,1). AUC is as the probability that the model ranks a random positive example more highly than a random negative example.

A screenshot of a cell phone

Description automatically generated

TP Rate

FP Rate

* **AUC is desirable** for the following two reasons:
* AUC is **scale-invariant.** It measures how well predictions are ranked, rather than their absolute values.
* AUC is **classification-threshold-invariant.** It measures the quality of the model's predictions irrespective of what classification threshold is chosen.
* However, both these reasons come with caveats, which may **limit the usefulness of AUC i**n certain use cases:
* **Scale invariance is not always desirable.** For example, sometimes we really do need well calibrated probability outputs, and AUC won’t tell us about that.
* **Classification-threshold invariance is not always desirable.** In cases where there are wide disparities in the cost of false negatives vs. false positives, it may be critical to minimize one type of classification error. For example, when doing email spam detection, you likely want to prioritize minimizing false positives (even if that results in a significant increase of false negatives). AUC isn't a useful metric for this type of optimization.
* **Prediction Bias:** Prediction bias is a quantity that measures how far apart those two averages are:
* **Note:** A good model will usually have near-zero bias. That said, a low prediction bias does not prove that your model is good. A really terrible model could have a zero-prediction bias. For example, a model that just predicts the mean value for all examples would be a bad model, despite having zero bias.
* **Note: "**Prediction bias" is a different quantity than [bias](https://developers.google.com/machine-learning/crash-course/descending-into-ml) (the *b* in wx + b).
* Possible root causes of prediction bias are:
* Incomplete feature set
* Noisy data set
* Buggy pipeline
* Biased training sample
* Overly strong regularization

A screenshot of text

Description automatically generated

(each point = average of 1000 points)

* Why are the predictions so poor for only part of the model? Here are a few possibilities:
* The training set **doesn't adequately represent** certain subsets of the data space.
* Some subsets of the data set are **noisier** than others.
* The model is **overly regularized**. (Consider reducing the value of lambda.)
* Z-Score is a sort of technique which is used to normalize datasets by converting each raw value (including the label) to its Z-score.

**Regularization:**

**L0 Regularization:** Sparse feature crosses may significantly increase feature space. It causes out of memory and noise coefficients (overfitting). You may use regularization which is the enemy of overfitting. However, regularization does not decrease model size and memory usage. One way is to zero out some of the weights that could save us RAM and can also potentially help us with overfitting. It would **penalize** the **count of non-zero coefficient values** in a model. But we have to be careful we don’t want to lose the coefficients; we just want to lose one that are sort of extra noisy. It’s not convex, it’s hard to optimize. Instead of L0 try **L1 Regularization:** we can use L1 regularization to encourage many of the uninformative coefficients in our model to be exactly 0.

A screenshot of a cell phone

Description automatically generated

A picture containing bird

Description automatically generated

* You can think of the derivative of L2 as a force that removes x% of the weight every time. As [Zeno](https://wikipedia.org/wiki/Zeno%27s_paradoxes#Dichotomy_paradox) knew, even if you remove x percent of a number *billions of times*, the diminished number will still never quite reach zero. (Zeno was less familiar with floating-point precision limitations, which could possibly produce exactly zero.) At any rate, L2 does not normally drive weights to zero.

You can think of the derivative of L1 as a force that subtracts some constant from the weight every time. However, thanks to absolute values, L1 has a discontinuity at 0, which causes subtraction results that cross 0 to become zeroed out. For example, if subtraction would have forced a weight from +0.1 to -0.2, L1 will set the weight to exactly 0. Eureka, L1 zeroed out the weight.

* **L2 vs. L1:**

A screenshot of a cell phone

Description automatically generated

* **L1 Zeros Out:**

A picture containing bird, flower

Description automatically generated

|  |  |
| --- | --- |
| **A close up of a logo  Description automatically generated** | **A close up of a logo  Description automatically generated** |
| **Linear Neural Network** | **Non-Linear Neural Network** |

* **Dropout:** Works by randomly “dropping out” units in a network for a single gradient step. When we apply dropout, what we’re saying is that with probability P we take a node and we essentially remove it from the network for a single gradient step. On different gradient steps, we repeat and we’ll take different nodes to drop out randomly.
* **Backpropagation Failures:**
* **Vanishing Gradients:** The gradients for the lower layers (closer to the input) can become very small. In deep networks, computing these gradients can involve taking the product of many small terms.

When the gradients vanish toward 0 for the lower layers, these layers train very slowly, or not at all.

**The ReLU** activation function can help **prevent vanishing gradients**.

* **Exploding Gradients:** If the weights in a network are very large, then the gradients for the lower layers involve products of many large terms. In this case you can have exploding gradients: gradients that get too large to converge.

**Batch normalization** and **lowering the learning rate** can help **prevent exploding gradients.**

* **Dead ReLU Units:** Once the weighted sum for a ReLU unit falls below 0, the ReLU unit can get stuck. It outputs 0 activation, contributing nothing to the network's output, and gradients can no longer flow through it during backpropagation. With a source of gradients cut off, the input to the ReLU may not ever change enough to bring the weighted sum back above 0.

**Lowering the learning rate** can help **keep ReLU units from dying**.

A screenshot of a cell phone

Description automatically generated

* **Dropout Regularization:** Yet another form of regularization, called **Dropout**, is useful for neural networks. It works by randomly "dropping out" unit activations in a network for a single gradient step. The more you drop out, the stronger the regularization:

0.0 = No dropout regularization.

1.0 = Drop out everything. The model learns nothing.

Values between 0.0 and 1.0 = More useful.

**Multi-Class Classification:**One vs. all provides a way to leverage binary classification. Given a classification problem with N possible solutions, a one-vs.-all solution consists of N separate binary classifiers—one binary classifier for each possible outcome. During training, the model runs through a sequence of binary classifiers, training each to answer a separate classification question. For example, given a picture of a dog, five different recognizers might be trained, four seeing the image as a negative example (not a dog) and one seeing the image as a positive example (a dog). That is:

1. Is this image an apple? No.
2. Is this image a bear? No.
3. Is this image candy? No.
4. Is this image a dog? Yes.
5. Is this image an egg? No.

This approach is fairly reasonable when the total number of classes is small but becomes increasingly inefficient as the number of classes rises.

We can create a significantly more efficient one-vs.-all model with a deep neural network in which each output node represents a different class. The following figure suggests this approach:

**A close up of text on a white background

Description automatically generated**

* **One logistic regression** output node in our model **for every possible class.**
* **Softmax:** Recall that [*logistic regression*](https://developers.google.com/machine-learning/crash-course/logistic-regression) produces a decimal between 0 and 1.0. For example, a logistic regression output of 0.8 from an email classifier suggests an 80% chance of an email being spam and a 20% chance of it being not spam. Clearly, the sum of the probabilities of an email being either spam or not spam is 1.0.

***Softmax***extends this idea into a multi-class world. That is, Softmax assigns decimal probabilities to each class in a multi-class problem. Those decimal probabilities must add up to 1.0. This additional constraint helps training converge more quickly than it otherwise would.

* **Softmax Options:**
* **Full Softmax** is the Softmax we've been discussing; that is, Softmax calculates a probability for every possible class.
* **Candidate sampling** means that Softmax calculates a probability for all the positive labels but only for a random sample of negative labels. For example, if we are interested in determining whether an input image is a beagle or a bloodhound, we don't have to provide probabilities for every non-doggy example.
* **One Label on One Observation vs. Many Labels on One Observation:**

**Softmax** assumes that each example is a member of exactly one class. So, you may **not use Softmax for “many classes in one image”**. (For example, there are lots of fruits on an image such as apple, banana, peach and pear.) **Instead of Softmax, you’ll have to use multiple logistic regressions.**

**Week 4**

* **Embeddings:** Embeddings, translate large sparse vectors into a lower-dimensional space that preserves semantic relationships. They map features to d-dimensional real-valued vector space whose dimension values determines similarity between classes. It helps cluster similar classes and so you can recommend a pleasurable movie to a user. Sometimes, we can look at the embeddings and assign semantic meanings to the dimensions, and other times we cannot. Often, each such dimension is called a latent dimension, as it represents a feature that is not explicit in the data but rather inferred from it.
* It is the distances between movies in the embedding space that are meaningful, rather than a single movie's values along any given dimension.
* **Categorical data** refers to input features that represent one or more discrete items from a finite set of choices. For example, it can be the set of movies a user has watched, the set of words in a document, or the occupation of a person. Categorical data is most efficiently **represented via sparse tensors**, which are tensors with very few non-zero elements.
* Sparse representations have a couple of problems that can make it hard for a model to learn effectively.
* **Why Embeddings?** Categorical data is represented by one-hot encoding hence sparse tensors hence a huge number of weights. A large number of weights causes further problems:
* **Size of Network:**
* **Amount of data**. The more weights in your model, the more data you need to train effectively.
* **Amount of computation**. The more weights, the more computation required to train and use the model. It's easy to exceed the capabilities of your hardware.
* **Lack of meaningful relations between vectors.**

A close up of a map

Description automatically generated

* **Principal Component Analysis (PCA):**[principal component analysis](https://wikipedia.org/wiki/Principal_component_analysis) (PCA) has been used to create word embeddings. Given a set of instances like bag of words vectors, PCA tries to find highly correlated dimensions that can be collapsed into a single dimension.
* A **static model** is trained offline. That is, we train the model exactly once and then use that trained model for a while.
* A **dynamic model** is trained online. That is, data is continually entering the system and we're incorporating that data into the model through continuous updates.
* **offline inference,** meaning that you make all possible predictions in a batch, using a MapReduce or something similar. You then write the predictions to an SSTable or Bigtable, and then feed these to a cache/lookup table.
* **online inference,** meaning that you predict on demand, using a server.
* Here are the pros and cons of offline inference:
* Pro: Don’t need to worry much about cost of inference.
* Pro: Can likely use batch quota or some giant MapReduce.
* Pro: Can do post-verification of predictions before pushing.
* Con: Can only predict things we know about — bad for long tail.
* Con: Update latency is likely measured in hours or days.
* Here are the pros and cons of online inference:
* Pro: Can make a prediction on any new item as it comes in — great for long tail.
* Con: Compute intensive, latency sensitive—may limit model complexity.
* Con: Monitoring needs are more intensive.

**Tensorflow:**

* TensorFlow, as the name indicates, is a framework to define and run computations involving tensors
* A tensor is a generalization of vectors and matrices to potentially higher dimensions.
* Colab is Google's version of [Jupyter Notebook](https://jupyter.org/).
* **Tensorflow vs Keras**

**TensorFlow** is an end-to-end open source platform for machine learning. It’s a comprehensive and flexible ecosystem of tools, libraries and other resources that provide workflows with high-level APIs.

**Keras**, on the other hand, is a high-level neural networks library which is running on the top of TensorFlow, CNTK, and Theano. Using Keras in deep learning allows for easy and fast prototyping as well as running seamlessly on CPU and GPU. This framework is written in Python code which is easy to debug and allows ease for extensibility.

There are several differences between these two frameworks. Keras is a neural network library while TensorFlow is the open source library for a number of various tasks in machine learning. TensorFlow provides both high-level and low-level APIs while Keras provides only high-level APIs.

SINAV SORULARI:

Test verilerine overfit olduğunda elde ettiğin sonuç(accuracy) çok iyi gibi görünebilir ama hiç karşılaşmadığı veriler(gerçek hayat verileri) üzerinde kötü performans gösterecektir. Örnek veriyorum, yarın sınavımız var ve sürekli hocanın derste işlediği örnekleri sabaha kadar sular seller gibi ezberledik(overfitting). Sınava girdik(daha önce karşılaşmadığımız veriler) ve gördük ki hoca soruyu ters çevirmiş, suya daldırmış vs. Çuvalladık. Oysa ki farklı kaynaklardan farklı örnekler üzerinde de çalışanlar(training with more data), önemsiz konuları çalşmayıp kafasını bunlarla doldurmayanlar(feature removing), yeteri kadar çalıştığını düşünüp uykusunu alanlar(early stopping), örneklerin farklı çözüm yöntemlerini de araştırıp öğrenmesini bunlarla harmanlayanlar(ensembling), çalışmaya başlamadan önemli konulara yüksek önemsiz konulara düşük puan verip vaktini konunun puanına göre harcayanlar (regularization) daha önce karşılaşmadığı sorular karşısında daha başarılı sonuçlar aldı. Diğer sorunuzda ise 4. soruyu örnek alalım, ev fiyatlarını tahmin edebilen bir algoritma yazıyoruz diyelim, önceki ev sahibinin cinsiyetinin fiyat ile alakası olmadığı için bu özelliği (feature) kaldırırsak (bkz:feature selection) hem zaman ve paradan tasarruf edip hem de modeli kompleks olmaktan biraz uzaklaştırdığımız için overfittingden de uzaklaşmış olacağız. 1 taşla 2 kuş vurmuş olacağız. Örnekleme biraz zorlama oldu, umarım faydası olmuştur.