# 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**.

- **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.

| Type of ML Problem | Description | Example |
|---|---|---|
| Classification | Pick one of N labels | Cat, dog, horse, or bear |
| Regression | Predict numerical values | Click-through rate |
| Clustering | Group similar examples | Most relevant documents (unsupervised) |
| Association rule learning | Infer likely association patterns in data | If you buy hamburger buns, you're likely to buy hamburgers (unsupervised) |
| Structured output | Create complex output | Natural language parse trees, image recognition bounding boxes |
| Ranking | Identify position on a scale or status | Search result ranking |

- 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)** = $(\text{observation} - \text{prediction})^2$

$$L_2 Loss = \sum_{(x,y)\in D} (y - prediction(x))^2$$

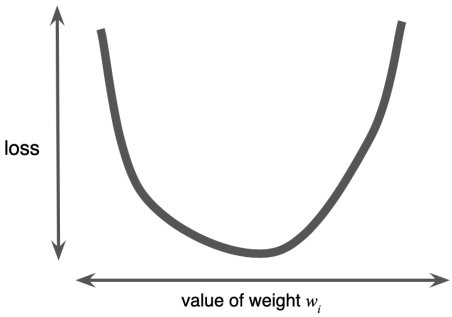
$\sum$ :We're summing over all examples in the training set.
$D$: Sometimes useful to average over all examples, so divide by $\|D\|$.

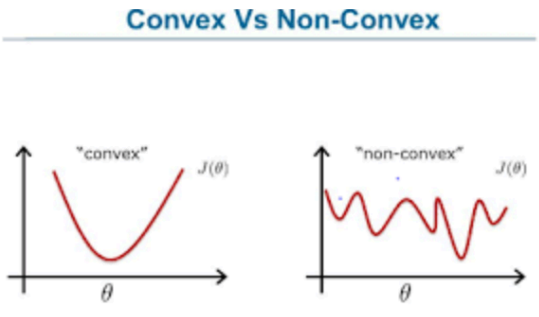
$$MSE = \frac{1}{N} \sum_{(x,y)\in D} (y - prediction(x))^2$$

- 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.



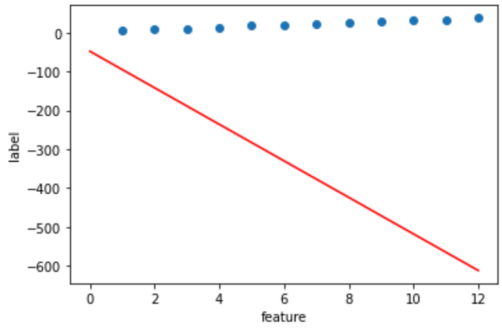
- Calculating the loss function for every conceivable value of $w_i$ over the entire data set would be an inefficient way of finding the convergence point. Let's examine a better mechanism—very popular in machine learning—called **gradient descent**.
- The **gradient** always points in the direction of steepest increase in the **loss** function.

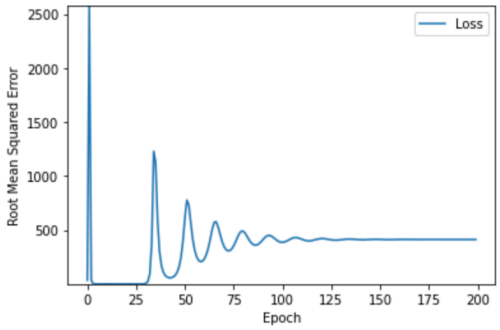
- In **gradient descent**, a **batch** is the total number of examples you use to calculate the gradient in a single iteration.
    - **Stochastic Gradient Descent / Online Gradient Descent:** Compute loss & gradients over one sample at a time.
    - **Batch Gradient Descent:** Compute loss & gradients 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.



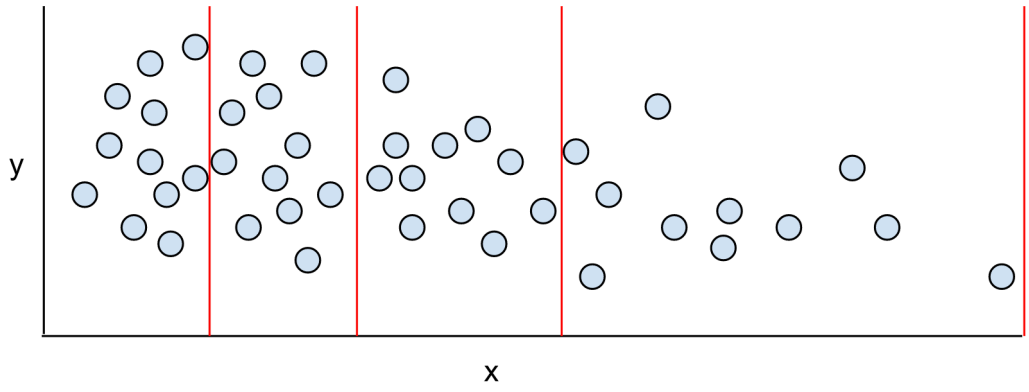

The resulting model is terrible; the red line doesn't align with the blue dots. Furthermore, the loss curve oscillates like a roller coaster. An oscillating loss curve strongly suggests that the learning rate is too high.

- **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:**

  quantile bucketing

  Distributing a feature's values into **buckets** so that each bucket contains the same (or almost the same) number of examples. For example, the following figure divides 44 points into 4 buckets, each of which contains 11 points. In order for each bucket in the figure to contain the same number of points, some buckets span a different width of x-values.

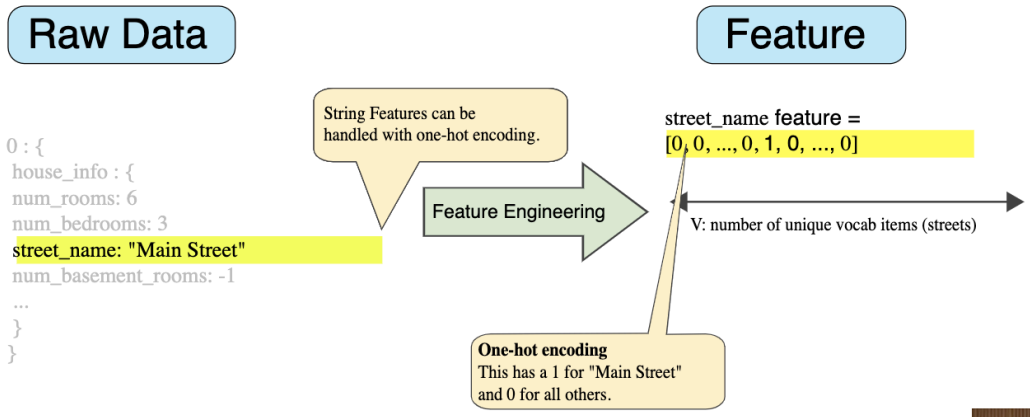



- An **overfit** model gets a low loss during training but does a poor job predicting new data. Overfitting is caused by making a model more complex than necessary.
- **generalization bounds:**
  - the complexity of the model
  - the model's performance on training data
- The following three basic assumptions 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.




  - Dictionary maps each street name to an int in `{0, ...,V-1}`
  - Now represent one-hot vector above as <*i*>

- 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.

-

However, if a user didn't enter a `quality_rating`, perhaps the data set represented its absence with a magic value like the following:

```
quality_rating: -1
```

✘

To explicitly mark magic values, create a Boolean feature that indicates whether or not a `quality_rating` was supplied. Give this Boolean feature a name like `is_quality_rating_defined`.

In the original feature, replace the magic values as follows:

- For variables that take a finite set of values (discrete variables), add a new value to the set and use it to signify that the feature value is missing.
- For continuous variables, ensure missing values do not affect the model by using the mean value of the feature's data.

## 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 (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.

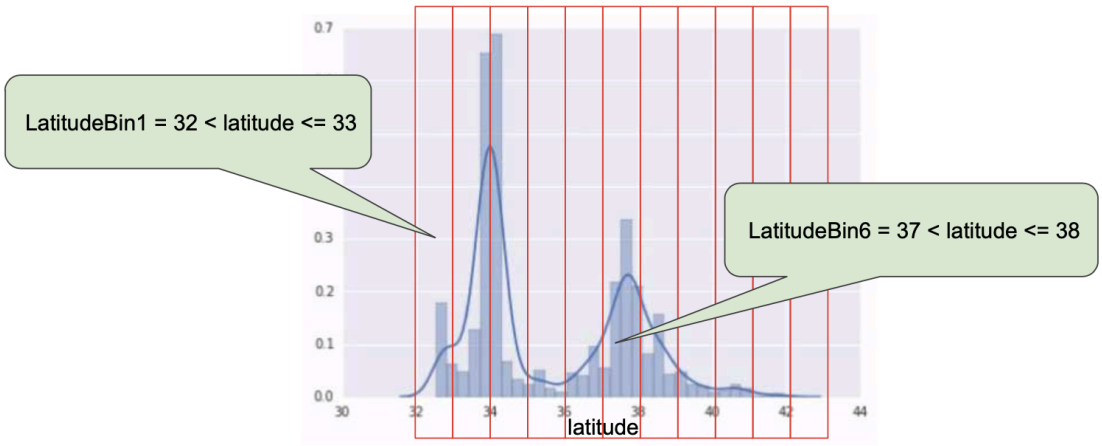



**Figure 8. Binning values.**

Instead of having one floating-point feature, we now have 11 distinct boolean features (`LatitudeBin1`, `LatitudeBin2`, ..., `LatitudeBin11`). Having 11 separate features is somewhat inelegant, so let's unite them into a single 11-element vector. Doing so will enable us to represent latitude 37.4 as follows:

```
[0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
```

- **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.) Let's create a feature cross named $x_3$ by crossing $x_1$ and $x_2$.

$$x_3 = x_1 + x_2$$

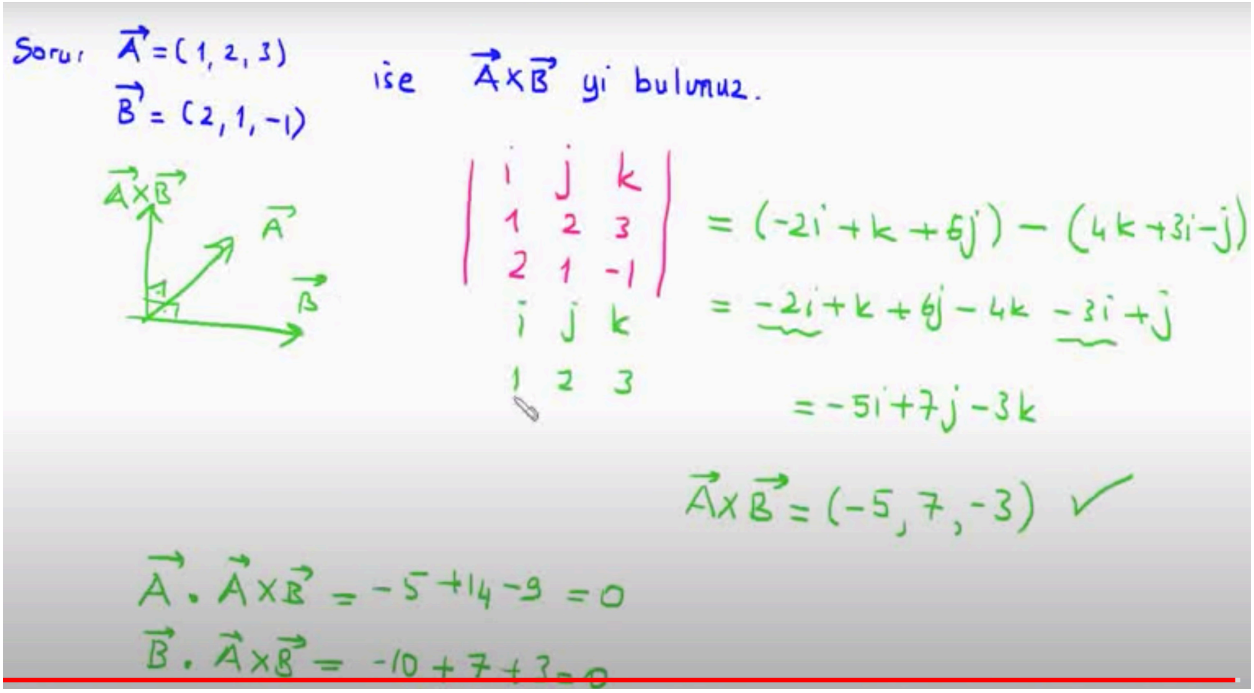
We treat this newly minted feature cross just like any other feature. The linear formula becomes:

$$y = b + w_1x_1 + w_2x_2 + w_3x_3$$

A linear algorithm can learn a weight for $w_3$ just as it would for $w_1$ and $w_2$. In other words, although $w_3$ encodes nonlinear information, you don't need to change how the linear model trains to determine the value of $w_3$.

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

- **Cross Product:**




- **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.

- Empirical Risk Minimization:
  - aims for low training error

$$\text{minimize: } Loss(Data \mid Model)$$

turn this

into this

- Structural Risk Minimization:
  - aims for low training error
  - while balancing against complexity

$$\text{minimize: } Loss(Data \mid Model) + complexity(Model)$$

## ♣ 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):

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:

$$L_2\ Regularization\ Term\ =\ ||w||^2\ =\ w_1^2\ +\ w_2^2 + \ldots + w_n^2$$

In this formula, weights close to zero have little effect on model complexity, while outlier weights can have a huge impact.

For example, a linear model with the following weights:

$$\{w_1 = 0.2, w_2 = 0.5, w_3 = 5, w_4 = 1, w_5 = 0.25, w_6 = 0.75\}$$

Has an $L_2$ regularization term of 26.915:

$$\begin{aligned} & w_1^2 + w_2^2 + \boldsymbol{w_3^2} + w_4^2 + w_5^2 + w_6^2 \\ &= 0.2^2 + 0.5^2 + \mathbf{5^2} + 1^2 + 0.25^2 + 0.75^2 \\ &= 0.04 + 0.25 + \mathbf{25} + 1 + 0.0625 + 0.5625 \\ &= 26.915 \end{aligned}$$

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 Loss Function with $L_2$ Regularization

$$Loss(Data|Model) + \lambda \left(w_1^2 + \ldots + w_n^2\right)$$

Where:

$Loss$: Aims for low training error
$\lambda$: Scalar value that controls how weights are balanced
$w_1^2 + \ldots + w_n^2$: Square of $L_2$ norm

- **How to choose L2 Regularization coefficient of Lambda ($\lambda$)?**
  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 $10^{-6}$ 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.
  -

Lowering the value of lambda tends to yield a flatter histogram, as shown in Figure 3.

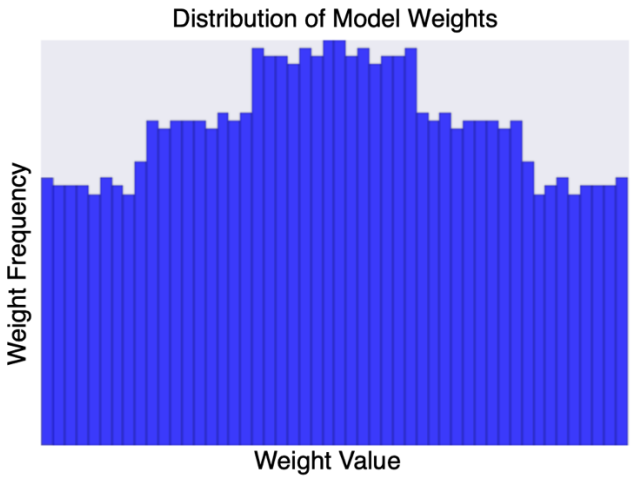



**Figure 3. Histogram of weights produced by a lower lambda value.**

- 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.

**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.

- **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.