Machine Learning Speciality

Learning plan

Courses

1. Udemy AWS Certified Machine Learning Speciality 2022 - Hands On!
2. Coursera Machine Learning Andrew Ng
3. Python for Data Visualization

Books and hands-on

1. DataScience on AWS
2. Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow: GitHub repository
3. Dive into Deep Learning (d2l) book
4. Python Data Science Handbook
5. Python Data Science Handbook notebooks
6. The Elements of Statistical Learning - Hastie et al
7. Practical Deep Learning for Coders by Sylvain Gugger and Jeremy Howard
8. Pattern recognition and machine learning by Bishop
9. Interpretable Machine Learning by Molnar

Resources

Certification

- AWS Certified Machine Learning - Speciality
- AI/ML Learning Journey
- AWS ML Specialty FabG GitHub repo

ML topics

- Bias-variation explanation
- 12 Types of Neural Network Activation Functions: How to Choose?
- Parametric and Nonparametric Machine Learning Algorithms
- Neural Network Activation Function Types

Feature engineering

- https://towardsdatascience.com/feature-engineering-for-machine-learning-3a5e293a5114
- https://machinelearningmastery.com/robust-scaler-transforms-for-machine-learning/
- 6 Powerful Feature Engineering Techniques For Time Series Data (using Python)
- Three Approaches to Encoding Time Information as Features for ML Models

Imputation

- DataWig - Imputation for Tables

Probability

- Probability Distributions and their Mass/Density Functions
- PROBABILISTIC APPROACHES: SCENARIO ANALYSIS, DECISION TREES AND SIMULATIONS
- Doing Bayesian Data Analysis - A Tutorial with R, JAGS, and Stan
- Probability Distribution Table - Intro with tossing a coin 3 times
- What is a Probability Distribution?
- Continuous Probability Distribution
- Khan Academy - Probability density function
- PennState STAT 414/415 - Probability Density Functions
- What is the relationship between the probability mass, density, and cumulative distribution functions?

https://quip-amazon.com/Zh52ATqZ1dbl/AWS-Certification-Prep-Machine-Learning-Specialty-MLS-C01

https://arxiv.org/pdf/1609.04836.pdf https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html https://academic.oup.com/bib/article/14/1/13/304457 https://www.omicsonline.org/open-access/a-comparison-of-six-methods-for-missing-data-imputation-2155-6180-1000224.php?aid=54590

https://docs.aws.amazon.com/sagemaker/latest/dg/object2vec.html https://aws.amazon.com/blogs/machine-learning/introduction-to-amazon-sagemaker-object2vec/ https://www.jair.org/index.php/jair/article/view/10302

- Coursera - Practical Data Science Specialization
- Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow: GitHub repository

I'm happy to report that I passed my AWS ML Specialty Certification test yesterday. A few points to share for those still on the journey:

- a CloudGuru and LinuxAcademy training should be considered a baseline only. You'll need to dive a good deal deeper on your own.
- Whizlabs practice tests (which have been instrumental in earning my other certs) aren't that well aligned with the actual test, they're much more verbose and there aren't enough of them (two full tests, 5 short subject-area tests), so if you use these to diagnose your weaknesses you won't have fresh tests by which to confirm you're on good ground.
- Groking the SageMaker Algorithm descriptions is critical, but requires a fair baseline in ML techniques. The above program's didn't give me enough, but by the time I finished with those below, and a fair bit of Internet reading on terms/techniques I didn't quite get, I started to get them. https://machinelearningmastery.com is a great resource.
- I found the Andrew NG Coursera program good for firming up my understanding of what's happening under the covers. I didn't go past the neural network classes.
- I also did the Google ML Crash course. Their playgrounds are great for getting intuition about tuning.
- I'd say that each of these courses built upon the others.
- Glue and the Kinesis suite are important to know, as is some EMR.
- I didn't get any questions where I needed to have memorized the exact names of hyperparameters (like in a lot of Whizlab questions), but be sure you know how learning rate, batchsize, epochs, dropout, the regularization params address, common training problems.

- Process of elimination is your friend, but often require good ML fundamentals to recognize the “obvious” wrong choices.

#aws-certified-machine-learning-speciality-exam slack channel

Udemy course

- AWS Certified Machine Learning Speciality 2022 - Hands On!
- Course materials

Data engineering

S3

Encryption:

- **SSE-S3**: keys handled and managed by AWS
- **SSE-KMS**: keys handled and managed by AWS KMS, audit trail for key usage, usage of CMKs
- **SSE-C**: customer manages own keys
- **Client Side Encryption**

Security:

- User based:
  - IAM policies
- Resource based:
  - bucket policies
  - ACL

new way of enforce encryption - use *S3 default encryption*

use VPC Endpoints to access S3 buckets

Logging & Audit:

- S3 access logs (stored in other S3 bucket)
- API calls logged in AWS CloudTrail

Tagged based security (combined with IAM policies and bucket policies)

EFS

- if your training data is already in Amazon Elastic File System (Amazon EFS), we recommend using that as your training data source
- EFS has the benefit of directly launching your training jobs from the service without the need for data movement, resulting in faster training start times

FSx for Lustre

- When your training data is already in Amazon S3 and you plan to run training jobs several times using different algorithms and parameters, consider using Amazon FSx for Lustre, a file system service

- The first time you run a training job, FSx for Lustre automatically copies data from Amazon S3 and makes it available to Amazon SageMaker. You can use the same Amazon FSx file system for subsequent iterations of training jobs, preventing repeated downloads of common Amazon S3 objects

Kinesis

- data is replicated to 3 AZ

Services:

- Kinesis Streams
- Kinesis Analytics
- Kinesis Firehose
- Kinesis Video Streams

Kinesis Data Streams (KDS)

- streams divided into Shards (Partitions)
- Shards have to be provisioned in advance (capacity planning)
- Data retention 24h default, up to 365d
- Record up to 1MB
- Data cannot be deleted

Capacity:

- Producer:
  - **1MB/s or 1000 msg/s at write per shard**
  - `ProvisionedThroughputException` otherwise
- Consumer classic:
  - 2MB/s at read per shard
  - 5 API calls/s per shard across all consumers

```
number of shards = max(inbound_write_bandwidth_in_MB,
outbound_read_bandwidth_in_MB/2)
```

inbound_write_bandwidth_in_MB and outbound_read_bandwidth_in_MB - across all producers/consumers

Kinesis Data Firehose (KDF)

Data transformation through Lambda function All or failed data can be saved to S3 backup bucket Automatic scaling Data conversion (CSV/JSON -> Parquet/ORC), only for S3 Supports compression for S3 (GZIP, ZIP, and SNAPPY) 60s minimum latency, 900s max buffer interval 1MB to 100MB buffer size pay per data amount delivery retry every 5s for 24h period 24h max retention can encrypt data at destination

each delivery stream:

- 2000 tps
- 5000 records per sec
- 5 MB per sec

Producers/data sources:

- Kinesis Data Streams
- CloudWatch, EventBridge
- AWS IoT
- AWS Pinpoint
- Kinesis Data Firehose API, Kinesis Agent
- Fluentbit

Destinations (batch writes):

- Amazon S3
- Amazon Redshift (copy through S3)
- Amazon OpenSearch
- HTTP endpoint
- 3rd party destinations (e.g. Splunk, Datadog, NewRelic, Dynatrace, Sumologic, MongoDB)

KDF vs. KDS

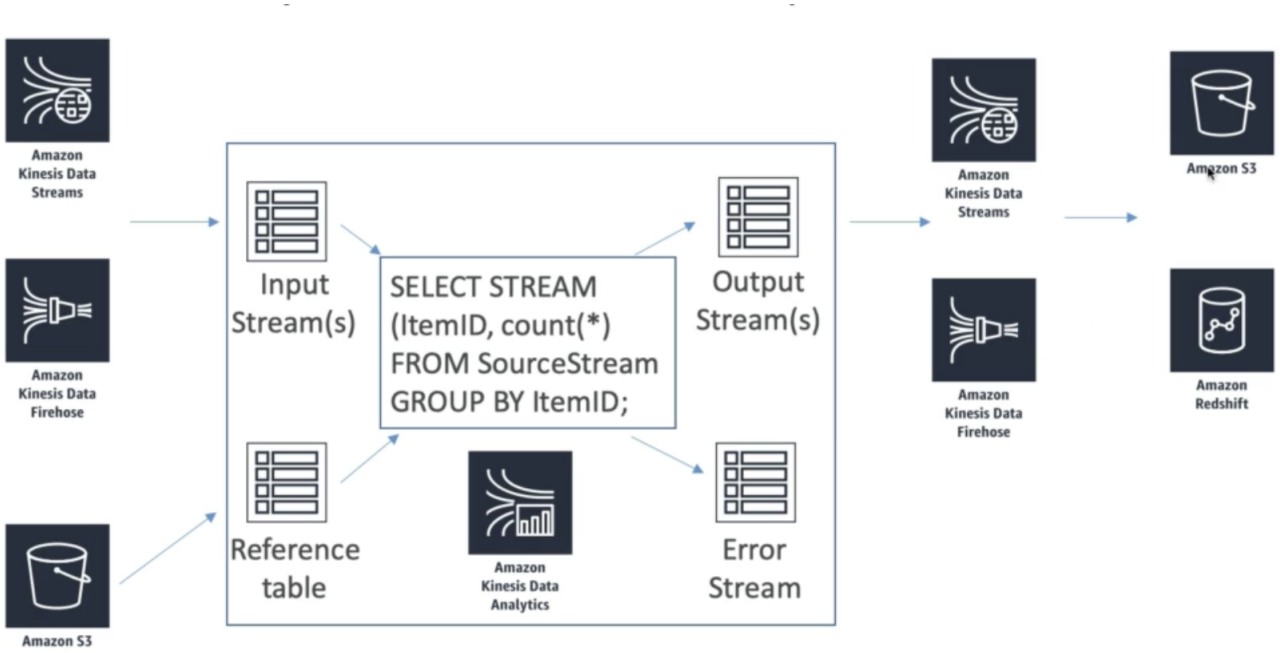
KDS:

- need to write custom code (producer/consumer) - KDS API
- real time, 200ms latency for classic, 70ms latency for enhanced fan-out
- **need to manage scaling** (shard splitting/merging)
- data storage 1 - 365d, replay capability, multi-consumers

KDF:

- Near real time (60s is lowest buffer time)
- Data transformation via Lambda
- **Auto scaling**
- no data storage, only delivery retry, 24h max retention

Kinesis Data Analytics (KDA)



Use cases:

- Streaming ETL
- Continuos metric generation
- Responsive analytics
- ML inference

SQL to Flink to write the computation Schema discovery

Inputs/streaming sources;

- Kinesis data stream
- Kinesis data firehose stream

If the data in your stream needs format conversion, transformation, enrichment, or filtering, you can preprocess the data using an AWS Lambda function. You can do this before your application SQL code executes or before your application creates a schema from your data stream.

Using a Lambda function for preprocessing records is useful in the following scenarios:

- Transforming records from other formats (such as `KPL` or `GZIP`) into formats that Kinesis Data Analytics can analyze. Kinesis Data Analytics currently supports JSON or CSV data formats.
- Expanding data into a format that is more accessible for operations such as aggregation or anomaly detection. For instance, if several data values are stored together in a string, you can expand the data into separate columns.
- Data enrichment with other Amazon services, such as extrapolation or error correction.
- Applying complex string transformation to record fields.
- Data filtering for cleaning up the data.

ML on KDA:

- **Random cut forest** (`RANDOM_CUT_FOREST` SQL function) for anomaly detection
- `HOTSPOTS` for detecting dense regions in the data

Kinesis Video Streams (KVS)

Data retention 1h - 10y

Producers:

- cameras, AWS DeepLens, radar data etc
- Producer SDK
- one producer per video stream

Consumers:

- AWS SageMaker
- Amazon Rekognition Video
- custom (MXNet, Tensorflow)

Use cases:

- consume video real-time + inference

Glue

Transformations:

- FindMatches ML: de-duplication

Glue ETL source connectors

AWS Data Stores

AWS Data Pipeline

Glue vs. Data Pipeline Glue:

- Run Apache Spark code, Scala or Python, focus on ETL
- Managed serverless
- Data catalog

Data Pipeline:

- Orchestration service
- More control over the environment, compute resources that run code & code
- Allow access to EC2 or EMR instances

AWS Batch

- Run batch jobs as Docker images, can run any computing job with provided Docker image
- Dynamic provisioning of the instances (EC2 & spot)
- Run workloads on AWS Fargate, EC2, Spot
- Optimal quantity and type based on volume and requirements
- Fully serverless
- Pay for underlying EC2 instances

- Schedule batch jobs using CWE
- Orchestrate using Step Functions

AWS DMS

AWS Step Functions

Max execution time 1y

EDA

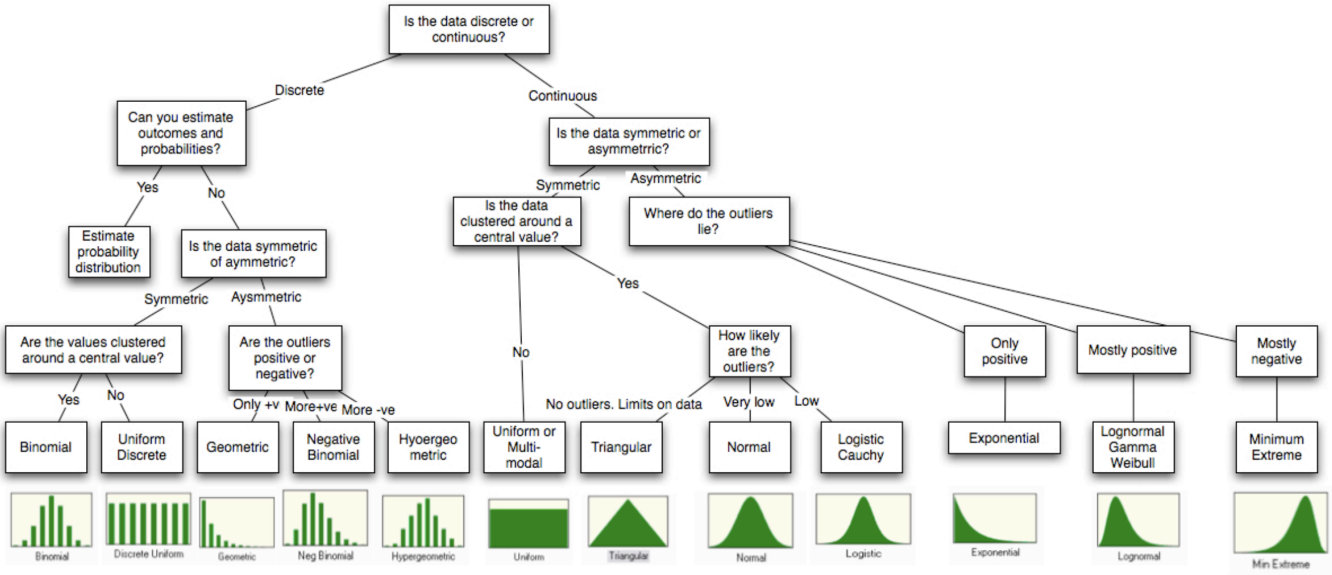
T-SNE T-SNE is a visualization technique for high dimensional datasets

For example, in natural language processing, we need to convert a word to a vector.

Types of data

- Numerical
- Categorical -> ordinal (ordered, e.g. size or rating), nominal (unordered, e.g. color)
- Text

Data distributions



Is data *discrete* or *continuous*?

- **probability density function (p.d.f.)** -> used to describe continuous probability distributions
- **probability mass function (p.m.f)** -> used to describe discrete probability distributions

E.g. continuous distribution:

- normal

E.g. discrete distribution:

- Poisson
- Binominal

- Bernoulli

Distribution types

Data types:

- discrete data
- continuous data

Bernoulli distribution

- has only **two** possible outcomes: 1 or 0 and a **single trial**
- independent trails
- probability of success (1) p, probability of failure (0) $1-p$
- probability mass function: $p^{x}(1-p)^{1-x}, x\in{(0,1)}$ or: [P(x) = \begin{cases} 1-p, & x=0\ p, & x=1 \end{cases}]
- expected value $E(X) = 1p + 0(1-p) = p$
- variance: $Var(X) = E(X^2)-[E(X)]^2 = p - p^2 = p(1-p)$

Uniform distribution

- n possible outcomes and probability of each outcome is equally likely
- density function: $f(x)=\frac{1}{b-a}$, a and b are parameters
- mean: $E(X)=(a+b)/2$
- variance: $Var(X)=(b-a)^2/12$

Binominal distribution

- only **two** possible outcomes
- each trial is independent
- a total number of n identical trials are conducted
- the probability of success and failure is **same** for all trials
- Bernoulli Distribution is a special case of Binomial Distribution with a **single trial**
- mathematical representation ($q = 1-p$) $$ P(x)=\frac{n!}{(n-x)!x!}p^xq^{n-x} $$
- mean: $\mu=np$
- variance: $Var(X)=np(1-p)=npq$

Normal distribution

- The mean, median and mode of the distribution coincide.
- The curve of the distribution is bell-shaped and symmetrical about the line x=μ.
- The total area under the curve is 1.
- Exactly half of the values are to the left of the center and the other half to the right
- mean: $E(x)=\mu$
- variance: $Var(X)=\sigma^2$, σ is standard deviation

Poisson distribution (discrete)

Poisson Distribution is applicable in situations where events occur at random points of time and space wherein our interest lies only in the number of occurrences of the event. E.g. models the entire number of calls at a call center in a day, number of printing error at each page of the book, number of customers arriving at a shop in an hour.

Parameter: λ - expected rate of occurrences

Following conditions must be met:

- Any successful event should not influence the outcome of another successful event
- The probability of success over a short interval must equal the probability of success over a longer interval
- The probability of success in an interval approaches zero as the interval becomes smaller

Exponential distribution

Exponential distribution models the interval of time between the calls in the call center (for example). Other examples:

- Length of time beteeen metro arrivals,
- Length of time between arrivals at a gas station
- The life of an air conditioner

Time Series

- Trends
- Seasonality
- Both
- Noise

Seasonality refers to periodic changes, while **trends** are longer-term changes over time. A trend across seasonal data would result in periodic seasonal spikes and valleys increasing or decreasing over time.

Additive model:

- seasonality + trends + noise seasonal variation is constant

Multiplicative model:

- seasonality x trends x noise seasonal variation increases as the trend increases

Amazon Athena

Amazon QuickSight

ML Features:

- Anomaly detection (RCF)
- Forecasting
- Auto-narratives

EMR & Hadoop

Managed Hadoop framework on EC2 instances
Spark, HBase, Presto, Hive, etc
EMR notebooks

Main node
Core node(s), can be scaled up and down
Task node(s), good use of spot instances

Can use AWS Data Pipeline to schedule and start/stop your cluster

EMR Storage:

- HDFS
- EMRFS
- EBS for HDFS
- Local file system

Apache Spark on EMR

Hadoop: MapReduce, YARN, HDFS Apache Spark: MapReduce, Spark, YARN, HDFS

Spark MLLib:

- classification, regression, DT, recommendation engine (ALS), clustering (K-Means), LDA (topic modeling), SVD, PCA, statistics, ML workflow utilities

EMR: choose instance types:

- Main node:
  `m4.large` for < 50 nodes, `m4.xlarge` > 50 nodes
- Core/Task nodes:
  `m4.large` is baseline, `m4.xlarge` for improved performance `t2.medium` if a cluster waits a lot for external dependencies

Amazon EMR node types

apache spark node types

- Master node on a spot instance:
  - only if you can tolerate sudden cluster termination
- Core nodes on spot instances:
  - core nodes process data and store information using HDFS
  - only if you can tolerate partial HDFS data loss
- Tasks nodes on spot instances:
  - task nodes process data but do not hold persistent data in HDFS
  - no data loss if terminated

Feature engineering

- Feature *construction*: multiplication, squaring, etc.
- Feature *extraction*: encoding, vectorization, etc.

- Feature *selection*: dimensionality reduction

Box plot

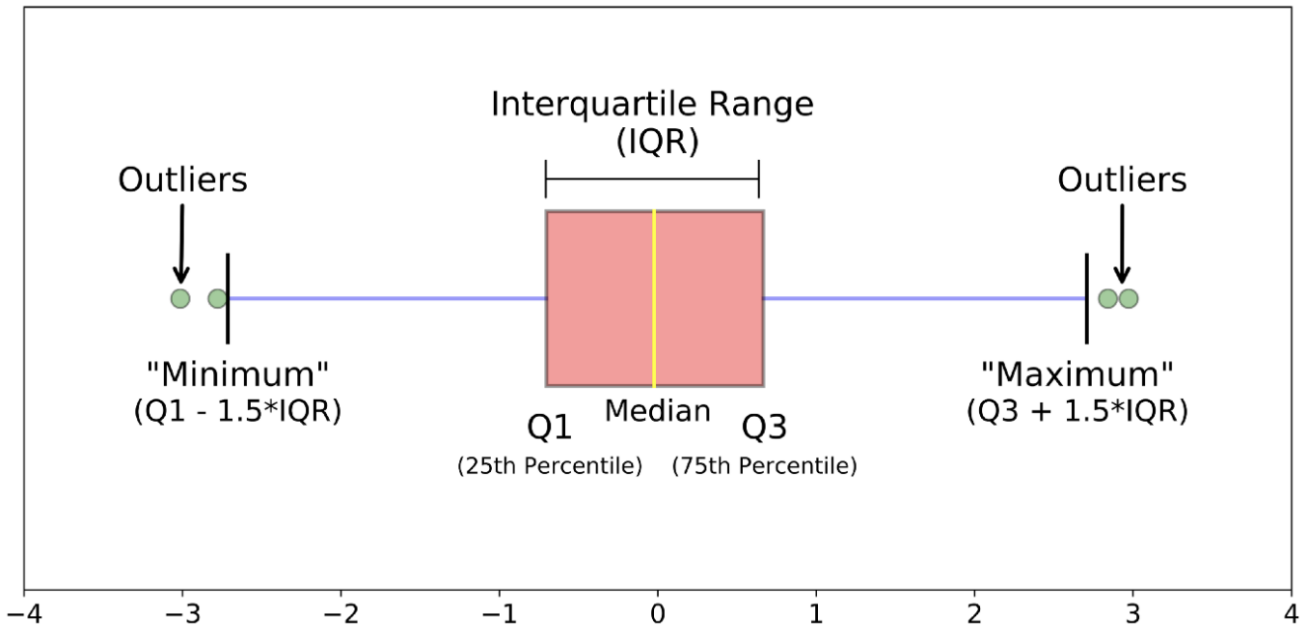
Display the data distribution based on five number summary:

- minimum = Q1 - 1.5 * IQR
- Q1
- median (Q2/50th percentile)
- Q3
- maximum = Q3 + 1.5 * IQR
- Interquartile Range (`IQR`) = Q3-Q1
- Minimum outlier cutoff = Q1 - 1.5 * IQR
- Maximum outlier cutoff = Q3 + 1.5 * IQR

For normal distribution:

- probability for value to be within IQR: 50%
- probability for value to be within [min, max]: 99.3%
- probability of an outlier (x < min or x > max): 0.7%

Understanding Boxplots



Imputing missing data

mean values
median is better when outliers are present

mode
placeholder - impute a constant value **most frequent** - works for categorical data

Limitations:

- only works on column level, misses correlations between features
- can't use mean/median/mode on categorical features (imputing with the most frequent value can work)
- not very accurate

dropping rows with missing data

- never going to be the right answer for "best" approach
- almost anything is better
- can lead to overfitting/underfitting

ML-based imputing:

- KNN: find K "nearest" (most similar) rows and average their values - works on *numerical* data

Better for *categorical* data:

- Deep learning
- Regression: find linear or non-linear relationship between the missing feature and other features
  most advanced: **MICE** - multiple imputation by chained equations

Better than imputing data - getting more real data! Try harder or collect more data

Addressing class imbalance

- **down-sampling**: reduce the size of the dominant class(es)
- **up-sampling**: increase the size of rare/small class(es), duplicate samples from the rare class(es)
- **data generation**: create new records, similar but not identical
- **sample weights**: for a model that uses a cost function, assign higher weights to rare classes

SMOTE Synthetic Minority Over-sampling Technique

- generates new samples of the rare class using KNN
- undersamples majority class
- better than just oversampling

Adjust thresholds

- increasing threshold will reduce false positive, but can result in more false negatives

Handling outliers

metrics are:

- **variance**: average of the squared differences from the mean
- **stdev**: sqrt(variance)

data points further than 1 stdev can be considered outliers

Dealing with outliers:

- remove
- use RCF (Random Cut Forest) for outlier detection
- use **Logarithm transformation**
- use **Robust standardization**

Encoding

categorical:

- Ordinal encoding
- One hot encoding
- Target (mean) encoding: average the target value for each category, replace the categorical values with the average target value.

Numerical:

- binning

Define a hierarchy structure

Incremental learning

Over time, you might find that a model generates inferences that are not as good as they were in the past. With incremental training, you can use the artifacts from an existing model and use an expanded dataset to train a new model. Incremental training saves both time and resources.

You can use incremental training to:

- Train a new model using an expanded dataset that contains an underlying pattern that was not accounted for in the previous training and which resulted in poor model performance.
- Use the model artifacts or a portion of the model artifacts from a popular publicly available model in a training job. You don't need to train a new model from scratch.
- Resume a training job that was stopped.
- Train several variants of a model, either with different hyperparameter settings or using different datasets.

`File` input mode only

- only three built-in algorithms support incremental training:
    - Object detection
    - Image classification
    - Semantic segmentation

Documentation

Binning

- Bucket observations together based on range of values
- Quantile binning ensures even sizes of bins (equal number of samples in each bin)

- Transforms numeric data to ordinal data
- Useful when there is uncertainty in the measurements

Transforming

- Apply some function to a feature to make it better suited for training
- e.g. `log()` or `sqrt()` etc

Scaling/Normalizing

- most models require feature data to at least be scaled to comparable values

Shuffling

SageMaker Ground Truth

GT creates its own model as images are labeled by people. As model learns, only images the model isn't sure about are sent to human labelers.

TF-IDF

Term Frequency - Inverse Document Frequency
Measure of how important and unique this word is for a specific document

Term Frequency (TF): how often a word occurs in a document **Document Frequency (DF)**: how often a word occurs in an entire set of documents

```
TF-IDF = TF/DF = TF*1/DF = TF*IDF
```

```
TF= number_of_term_occurences_in_document/total_number_of_words_in_document
IDF=number_of_documents_include_term/total_number_of_documents
```

Practically the `log(IDF)` is used -> better weighting of a word's overall popularity

compute relevancy for `n-grams` as well

Compute TF-IDF for unigrams and bigrams for 2 documents:

	I	Love	Certification	Exams	Puppies	I love	Love certification	Love puppies	Certification exams
"I love certification exams"									
"I love puppies"									

Metrics

Accuracy
ratio of cases classified *correctly*

$accuracy = \frac{TP+TN}{TP+TN+FP+FN}$

Precision
accuracy of predicted positive, ignores TN (!) AKA correct positives

- How many predicted items are relevant?

$precision = \frac{TP}{TP+FP}$

Recall
ability to predict a positive outcome
AKA sensitivity, true positive rate, completeness

- How many relevant items are predicted?

$recall = \frac{TP}{TP+FN}$

Specificity
True negative rate

$specificity = \frac{TN}{TN+FP}$

False Positive Rate (FPR) measures the false alarm rate or the fraction of actual negatives that are predicted as positive.

$FPR=\frac{FP}{FP + TN}$

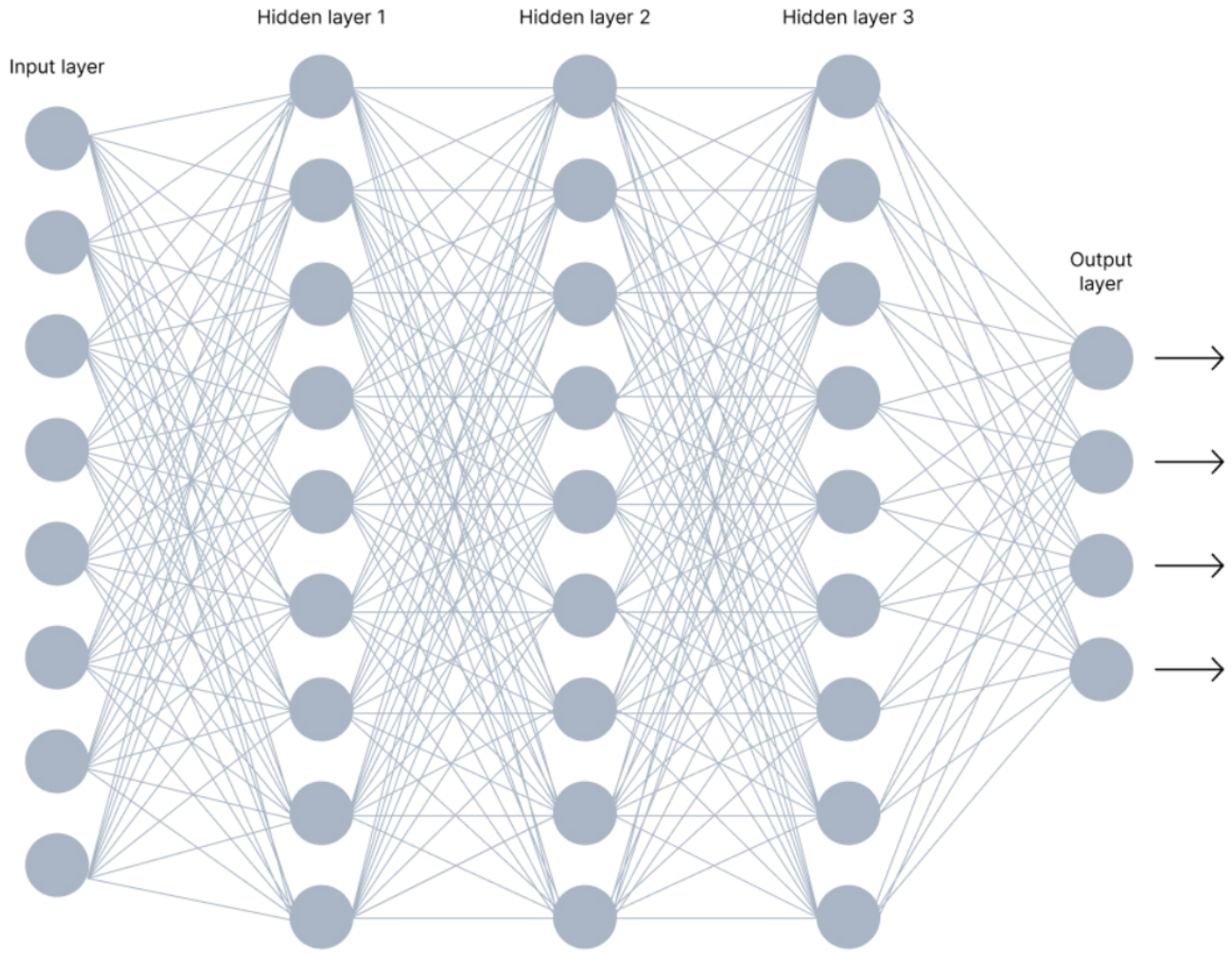
F1 Score
Harmonic mean of precision and sensitivity

$F1 = \frac{2TP}{2TP+FP+FN}$

RMSE
Root mean square error only cares about right/wrong answers

AUC ROC AUC 0.5 is useless, 1.0 is perfect

Modeling: Deep Learning



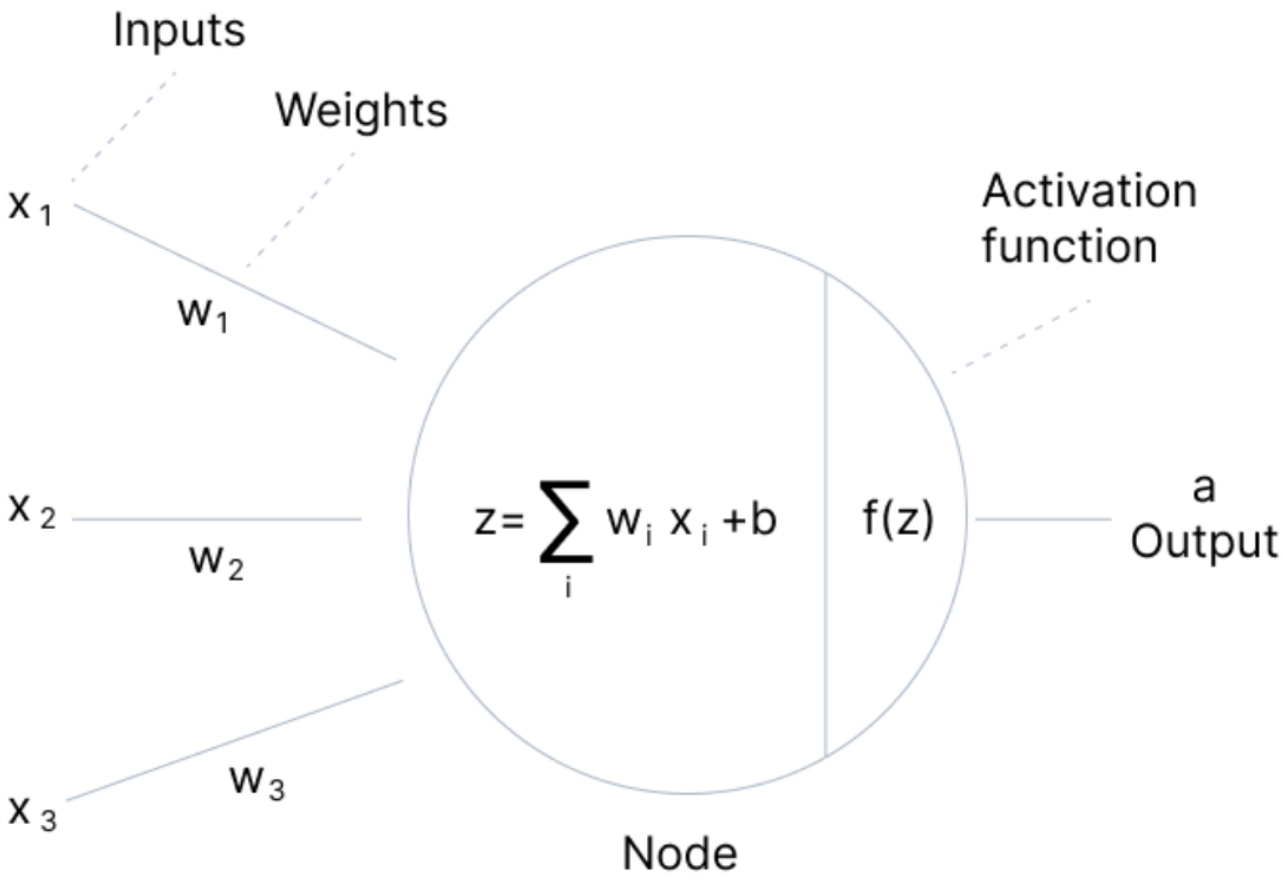
Input layer:
The input layer takes raw input from the domain. No computation is performed at this layer. Nodes here just pass on the information (features) to the hidden layer.

Hidden layer:
The hidden layer performs all kinds of computation on the features entered through the input layer and transfers the result to the output layer.

Output Layer:
It's the final layer of the network that brings the information learned through the hidden layer and delivers the final value as a result.

Note: All hidden layers usually use the same activation function. However, the output layer will typically use a different activation function from the hidden layers. The choice depends on the goal or type of prediction made by the model.

Each neuron is characterized by its **weight**, **bias**, and **activation function**:



Feedforward Propagation - the flow of information occurs in the forward direction. The input is used to calculate some intermediate function in the hidden layer, which is then used to calculate an output.

Backpropagation - the weights of the network connections are repeatedly adjusted to minimize the difference between the actual output vector of the net and the desired output vector.

Backpropagation aims to minimize the cost function by adjusting the network's weights and biases. The cost function gradients determine the level of adjustment with respect to parameters like activation function, weights, bias, etc.

DL frameworks:

- Tensorflow/Keras
- MXNet

Types of neural networks:

- Feedforward Neural Network
- Convolutional NN (CNN), e.g. for image classification
- Recurrent NN (RNN), e.g. for predicting sequences in time such as stock prices, translation, words in sentence
    - LSTM, GRU

Activation functions

- Define the output of a node/neuron given its input signal
- Help the neural network to use important information while suppressing irrelevant data points
- without an activation function, every neuron will only be performing a linear transformation on the inputs using the weights and biases. All layers will behave in the same way because the composition of two linear functions is a linear function itself. Although the neural network becomes simpler, learning any complex task is impossible, and our model would be just a linear regression model.
- A neural network will almost always have the same activation function in all hidden layers. This activation function should be differentiable so that the parameters of the network are learned in backpropagation
- ReLU is the most commonly used activation function for hidden layers
- While selecting an activation function, you must consider the problems it might face: vanishing and exploding gradients
- Regarding the output layer, we must always consider the expected value range of the predictions. If it can be any numeric value (as in case of the regression problem) you can use the linear activation function or ReLU
- Use Softmax or Sigmoid function for the classification problems

2 linear NN activation functions:

- *Linear*: not possible to use backpropagation as the derivative is constant (no relation to x), all layers degenerate to a single layer (all linear functions). Is simply a linear regression model
- *Binary step*: cannot be used for multi-class classification problems, the gradient = 0, causes a hindrance in the backpropagation process

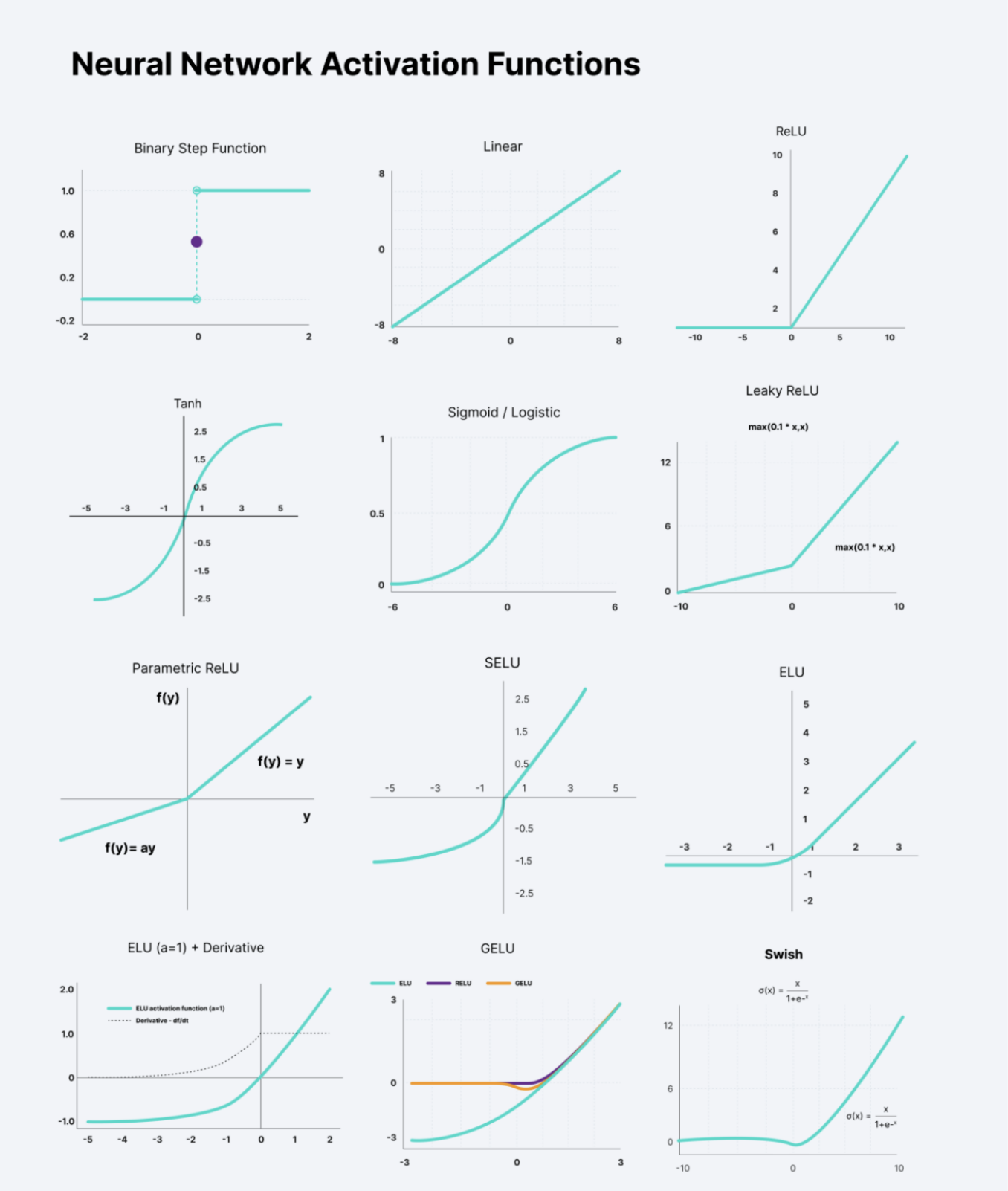
Non-linear activation functions:

- can create **complex mappings** (non-linear) between inputs and outputs
- allow **backpropagation** (because they have a useful derivative)
- allow for **multiple layers** (linear functions degenerate to a single layer)

10 non-linear NN activation functions:

- *Sigmoid (Logistic)* -> scales (0,1)
  - commonly used for models where we have to predict the probability as an output
  - differentiable and provides a smooth gradient
  - limitation: gradient values are only significant for range (-3,3) (vanishing gradient problem)
- *TanH (Hyperbolic Tangent)* -> scales (-1, 1)
  - output is zero-centered, output values can be easily mapped as strongly negative, neutral, or strongly positive
  - usually used in hidden layers of a neutral network - mean for the hidden layer comes out to be 0 or very close to; makes learning for the next layer much easier
  - also faces the vanishing gradient problem
  - 💡 Tanh generally preferred over sigmoid
  - Compute expensive
- *Rectified Linear Unit (ReLU)* `f(x) = max(0,x)`
  - has a derivative and allows for backpropagation
  - main characteristic is that ReLU doesn't activate all the neurons at the same time
  - computationally efficient compared to sigmoid and tanh
  - accelerates the convergence of gradient descent towards the global minimum of the loss function

    - has "**Dying ReLU problem**": All the negative input values become zero immediately, which decreases the model's ability to fit or train from the data properly.
- *Leaky ReLU* `f(x) = max(0.1*x,x)`: improved version to solve the Dying ReLU problem (has a small positive slope in the negative area)
    - enables backpropagation even for negative input values
    - limitation: the gradient for negative values is a small value that makes the learning of model parameters time-consuming
    - limitation: the predictions may not be consistent for negative input values
- *Parametric ReLU* `f(x) = max(a*x,x)`
- *Exponential Linear Unit (ELU)*: variant of ReLU that modifies the slope of the negative part of the function
    - strong alternative for ReLU
    - becomes smooth slowly until its output equal to -α whereas RELU sharply smoothes
    - avoids dead ReLU problem by introducing log curve for negative values of input
    - computationally expensive
    - no learning of α
    - exploding gradient problem
- *Softmax*
    - used for the final output layer, converts outputs to probabilities of each classification
    - it calculates the relative probabilities. Similar to the sigmoid/logistic activation function, the SoftMax function returns the probability of each class
    - most commonly used as an activation function for the last layer of the neural network in the case of multi-class classification
- *Swish* `f(x) = x*sigmoid(x)`
    - consistently matches or outperforms ReLU activation function on deep networks for image classification, machine translation etc
    - smooth function that means that it does not abruptly change direction like ReLU does near x = 0
    - Small negative values were zeroed out in ReLU activation function. However, those negative values may still be relevant for capturing patterns underlying the data. Large negative values are zeroed out for reasons of sparsity making it a win-win situation
    - being non-monotonous enhances the expression of input data and weight to be learnt
- *Gaussian Error Linear Unit (GELU)*
    - This activation function is motivated by combining properties from dropout, zoneout, and ReLUs
    - GELU nonlinearity is better than ReLU and ELU activations and finds performance improvements across all tasks in domains of computer vision, natural language processing, and speech recognition
- *Maxout*



How to choose the right activation function

You need to match your activation function for your output layer based on the type of prediction problem that you are solving—specifically, the type of predicted variable.

- For multi-class, use softmax on the output layer
- RNN's do well with Tanh
- For everything else:
    - start with ReLU

    - ReLU should only be used in the hidden layers
    - if you need to do better, try Leaky ReLU/Parametric ReLU
    - last resort: PReLU, Maxout
    - Swish for really deep (> 40 layers) networks
    - Sigmoid if you need more than one classification for same thing
    - Sigmoid/Logistic and Tanh functions should not be used in hidden layers as they make the model more susceptible to problems during training (due to vanishing gradients)

Based on type of prediction problem, use for **output** layer:

- **Regression** - Linear Activation Function
- **Binary Classification** - Sigmoid/Logistic Activation Function
- **Multiclass Classification** - Softmax
- **Multilabel Classification** - Sigmoid

Based on the type of NN architecture, use for **hidden** layers:

- **Convolutional Neural Network (CNN)**: ReLU activation function
- **Recurrent Neural Network (RNN)**: Tanh and/or Sigmoid activation function

CNN

- "feature-location invariant"
- can find features that aren't in a specific spot
- images, translation, sentence classification, sentiment analysis

Typical usage:
Conv2D > MaxPooling2D > Dropout > Flatten > Dense > Dropout > Softmax

Specialized CNN architectures:

- Define specific arrangement of layers, padding, and hyperparameters E.g.:
- LeNet-5
- AlexNet
- GoogLeNet
- ResNet

RNN

- Time-series data:
    - web logs, sensor logs, stock trades
    - self-driving based on past trajectories
- Data that consists of sequences of arbitrary length:
    - machine translation
    - image captions
    - machine-generated music

Concept: a recurrent neuron > a "memory cell", maintain memory about previous behavior, layer of recurrent neurons

RNN topologies

- Sequence to sequence: e.g. predict stock prices
- Sequence to vector: e.g. words in a sentence to sentiment
- Vector to sequence: e.g. create a caption from an image
- Encoder to decoder: e.g. sequence > vector > sequence, machine translation

Training:

- backpropagation through time

LSTM Cell:

- Long Short-Term Memory Cell
- maintains separate short-term and long-term states

GRU Cell:

- Gated Recurrent Unit
- Simplified LSTM Cell that performs about as well

Deep learning on EC2/EMR

- EMR supports MXNet and GPU instance types

Appropriate instance types:

- P3: up to 8 Tesla V100 GPUs
- P2: up to 16 K80 GPUs
- G3: up to 4 M60 GPUs
- Deep learning AMI

Tuning neural networks

Learning rate:

- NN are trained by gradient descent (or similar means)
- minimize some cost function over many *epochs*
- *learning rate*: how far apart these samples

Learning rate is a **hyperparameter**

- Too high -> you might overshoot the optimal solution
- Too small -> it might take too long to find the optimal solution

Batch size: how many training samples are used withing each epoch

- Smaller batch sizes can work their way out of local minima more easily
- Too large batch sizes can get stuck in the wrong solution
- Random shuffling at each epoch can make this look like very inconsistent results from run to run with large batch size

Important:

- Small batch sizes tend to **not** get stuck in local minima
- Small batch size have **regularization** effect
- Large batch sizes can converge on the wrong solution at random (large batch sizes tend to get stuck, at random, inside "local minima" instead of the correct solution)
- Large learning rates can overshoot the correct solution
- Small learning rates can increase training time

Regularization techniques for NN

- Prevents *overfitting*
- Used to balance model complexity and model fit, with the ultimate goal of ensuring the model generalize beyond the training dataset
- Compromise between fit and complexity (drop features, reduce weights)

Too many layers or too many neurons

Regularization:

- use simpler model, **less** layers, **less** neurons
- drop features, reduce weights
- **Dropout** - remove some neurons
- **Early stopping**
- Tune model complexity by adding a *penalty* for complexity to the cost function `C(x)`: $$ C_{reg}(w) = C(w) + \alpha*penalty(w) $$ α - a *regularizer* parameter

L1 and L2 regularization

Prevents overfitting in ML in general
A regularization term is added as weights are learned

- **L1** (LASSO): penalizes the model by sum of the abs(weights)
    - **performs feature selection** - entire features go to 0
    - essential if dimensionality is high due to feature engineering
    - computationally inefficient
    - sparse output, can make up for computational inefficiency
    - reduces the amount of noise in the model
    - feature selection can reduce dimensionality, out of 100 features, maybe only 10 end up with non-zero coefficients!
- **L2** (Ridge): penalizes the model by minimizing the sum of the squared weights
    - all features remain considered, just weighted
    - will try to make all weights smaller, but not necessarily driving them to 0
    - stabilizes the weights when there is high correlation between the input features
    - computationally efficient
    - dense output
    - better choice than L1 if all features are important
- **ElasticNet**: both L1 and L2

! Need to scale the features first !

L1 and L2 Regularization Methods

Vanishing gradient problem

Like the sigmoid function, certain activation functions squish an ample input space into a small output space between 0 and 1.

Therefore, a large change in the input of the sigmoid function will cause a small change in the output. Hence, the derivative becomes small. For shallow networks with only a few layers that use these activations, this isn't a big problem.

However, when more layers are used, it can cause the gradient to be too small for training to work effectively.

- when the slope of the learning curve approaches zero, things can get stuck
- becomes a problem with deeper networks and RNN's as these "vanishing gradients" propagate to deeper layers

Handling the vanishing gradient problem:

- Multi-level hierarchy, break up levels into their own sub-networks trained individually
- Long short-term memory (LSTM)
- Residual networks (e.g. ResNet)
- Ensemble of shorter networks
- Better choice of activation function, ReLU is a good choice

Exploding gradient problem

Exploding gradients are problems where significant error gradients accumulate and result in very large updates to neural network model weights during training.

An unstable network can result when there are exploding gradients, and the learning cannot be completed.

The values of the weights can also become so large as to overflow and result in something called NaN values.

Ensemble learning

Common example: random forest (RF)

- make lots of decision trees and let them all vote on the result - RF

Bagging:

- generate N new training sets by random sampling with replacement, each resampled model can be trained in **parallel**.

Boosting: Boosting is an ensemble method that creates a strong model by combining the predictions of multiple weak models.

- observations are weighted
- some will take part in new training sets more often

- training is **sequential**, each classifier takes into account the previous one's success - attempts to boost performance by reducing the errors of the previous model
- weak models need to be different
- using same data, same algorithm, but focus on **minimize the prediction error** -
- residual of the previous model (prediction error) - the data samples with the largest prediction error - becomes a **target** of the next model
- adjust next model's hyperparameter accordingly
- add each weak learner to the ensemble

Depending on your goals:

- **Boosting** generally yields better accuracy
- **Bagging** avoids overfitting
- Bagging is easier to parallelize

Modeling: Amazon SageMaker

ideal format - RecordIO/Protobuf

Training options:

- Built-in training algorithms
- Spark MLLib
- Custom Python Tensoflow/MXNet code
- Own custom docker image
- Algorithm from AWS marketplace

Model endpoint capabilities:

- On-demand inference (persistent endpoint)
- Batch transform
- Inference pipelines for complex processing
- SageMaker Neo for deploying to edge devices
- Elastic inference for accelerating deep learning models
- Automatic scaling for the endpoint
- The existing endpoint configuration is read-only

Built-in algorithms

- no coding required to start running experiments
- algorithms come with parallelization across multiple compute instances and GPU support:
    - algorithms that are parallelizable can be deployed on multiple compute instances for distributed training

Commonly supported `ContentType` values and the algorithms that use them:

ContentTypes for Built-in Algorithms

ContentType	Algorithm
application/x-image	Object Detection Algorithm, Semantic Segmentation
application/x-recordio	Object Detection Algorithm
application/x-recordio-protobuf	Factorization Machines, K-Means, k-NN, Latent Dirichlet Allocation, Linear Learner, NTM, PCA, RCF, Sequence-to-Sequence
application/jsonlines	BlazingText, DeepAR
image/jpeg	Object Detection Algorithm, Semantic Segmentation
image/png	Object Detection Algorithm, Semantic Segmentation
text/csv	IP Insights, K-Means, k-NN, Latent Dirichlet Allocation, Linear Learner, NTM, PCA, RCF, XGBoost
text/libsvm	XGBoost

- Many Amazon SageMaker algorithms support training with data in CSV format. To use data in CSV format for training, in the input data channel specification, specify `text/csv` as the `ContentType`.
- Amazon SageMaker requires that a CSV file does not have a header record and that the target variable is in the first column.
- To run unsupervised learning algorithms that don't have a target, specify the number of label columns in the content type. For example `content_type=text/csv;label_size=0`.

Supervised learning

General purpose algorithms that can be used for classification or regression problems:

- **Linear learner**: learns a linear function

- **Factorization machines**
- **XGBoost**
- **k-NN**: non-parametric

Specialized algorithms:

- **Object2Vec**
- **DeepAR forecasting**

Unsupervised learning

- **PCA**: reduces the dimensionality (number of features) within a dataset by projecting data points onto the first few principal components
- **K-Means**
- **IP insights**
- **Random Cut Forest (RCF)**: anomaly detection

Textual analysis

- **BlazingText**:
- **Sequence-to-Sequence**
- **Latent Dirichlet Allocation (LDA)**: determining topics in a set of documents, *unsupervised*
- **Neural Topic Model (NTM)**: unsupervised technique for determining topics in a set of documents, using a neural network approach.

Image processing

- **Image classification**
- **Semantic segmentation**
- **Object detection**

Links

- Common Information About Built-in Algorithms
- Use Amazon SageMaker Built-in Algorithms

Linear learner

- linear regression
- can handle both numeric regression and classification
- can do binary or multi class
- uses *softmax* loss function to train multiclass classifiers
- `File` or `Pipe` mode both supported
- supports `recordIO-wrapped protobuf` and `CSV` formats
- train on single or multi-machine CPU or GPU instances
- Parallelizable
- `dtype` of all feature and label values must be `float32`
- `CSV`: first column has to be the label
- the best model optimizes either of the following:

    - **Continuous objectives**, such as mean square error, cross entropy loss, absolute error.
    - **Discrete objectives** suited for classification, such as F1 measure, precision, recall, or accuracy.

How linear learner works **Step 1: Preprocessing**

- Training data must be **normalized** (can be done automatically by LL) -> mean = 0 and stddev = 1
    - use `normalize_data` and `normalize_label`
- Input data should be **shuffled** -> **That's important for algorithms trained using stochastic gradient descent**

Step 2: Training

- uses stochastic gradient descent (SGD)
- choose an optimization algorithm (Adam, AdaGrad, SGD, etc)
- multiple models are optimized in parallel
- L1, L2 regularization

Step 3: Validation

- validate and set the threshold
- most optimal model is selected -> is the one that achieves the best loss on the validation set

Model tuning

- by default, the linear learner algorithm tunes hyperparameters by training multiple models in parallel
- automatic model tuning (SageMaker) -> internal tuning mechanism is turned off automatically, `num_models` is set to 1

Hyperparameters

- `balance_multiclass_weights`: gives each class equal importance in loss functions
- `learning_rate`
- `mini_batch_size`
- `l1`: L1 regularization parameter
- `wd`: weight decay (L2 regularization)
- `loss`: specifies the loss function

Links

- Documentation
- Build multiclass classifiers with Amazon SageMaker linear learner
- Build a model to predict the impact of weather on urban air quality using Amazon SageMaker

XGBoost

- boosted group of decision trees
- uses gradient descent to minimize loss as new trees are added
- can be used for classification and regression (using regression trees)
- can be used as **framework** (**script** mode) (`from sagemaker.xgboost.estimator import XGBoost`) to run your customized training script that can incorporate additional data processing into

your training job. Example
- can be used as a built-in **algorithm** (`sagemaker.estimator.Estimator`) with XGBoost training container and `image_uris.retrieve()`
- takes `CSV`/`libsvm` or `Parquet`
- for `CSV`: the label is in the first column and CSV doesn't have a header record
- for `libsvm`: the label is in the first column. Subsequent columns contain the zero-based index value pairs for features
- CPU and GPU (for 1.2-1, single GPU instance)
- 1.0-1 or earlier - only CPU
- Parallelizable
- `File` and `Pipe` training input mode
- can be used in framework mode within notebooks (`sagemaker.xgboost`) or as built-in SageMaker algorithm
- **memory-bound**, not compute-bound
- uses CPU's only for multiple instance training
- a general-purpose instance (e.g. `m5`) is a good choice, better than a compute-optimized, e.g. `c4`
- v1.2 can train single-instance GPU, e.g. `p3`. `tree_method` hyperparameter must be set to `gpu_hist`

❗ For `CSV` training input mode, the total memory available to the algorithm (`Instance Count` * the memory available in the `InstanceType`) must be able to hold the training dataset. For `libsvm` training input mode, it's not required, but recommended.

Hyperparameters

Important:

- `subsample`: prevents overfitting
- `eta`: step size shrinkage, prevents overfitting
- `gamma`: minimum loss reduction to create a partition, larger = more conservative
- `alpha`: L1 regularization term, larger = more conservative
- `lambda`: L2 regularization term, larger = more conservative
- `eval_metric`: metric to optimize, e.g. AUC, error, mse
- `scale_pos_weight`: adjust balance of positive and negative weights, helpful for unbalanced classes, might be set to `sum(negative_classes)/sum(positive_classes)`
- `max_depth`: controls overfitting/underfitting
- `csv_weights`: allows to differentiate the importance of labelled data points by assigning each instance a weight value.

Links

- Documentation
- Multiclass classification with Amazon SageMaker XGBoost algorithm

Seq2Seq (Sequence-to-Sequence)

Supervised learning algorithm where the input is a sequence of tokens (for example, text, audio) and the output generated is another sequence of tokens. Applications:

- machine translation
- text summarization (input a longer string of words and predict a shorter string of words)
- speech-to-text

Uses RNNs and CNNs models with attention as encoder-decoder architectures

- expects data in `RecordIO-Protobuf` format
- tokens are expected as integers and not floats as majority of algorithms
- start with tokenized text files
- convert to `protobuf`
- three channels are required: `train`, `validation`, `vocab`
- **only supported on GPU instances and only train on a single instance, but multiple GPUs**
- can optimize on:
    - accuracy vs. provided validation dataset
    - `BLEU` score (compares against multiple reference translations)
    - Perplexity (cross-entropy)

Hyperparameters

- `batch_size`
- `optimizer_type`
- `learning_rate`
- `num_layers_encoder`
- `num_layers_decoder`

Links

- Documentation
- Example notebook
- Context-sensitive Spell Correction with Deep Learning

Deep AR

- forecast **scalar** (one-dimensional) time series data
- uses RNN's
- allows to train the same model over several related time series -> when your dataset contains hundreds of related time series, DeepAR outperforms the standard ARIMA and ETS methods -> "global" algorithm
- finds frequencies and seasonality
- doesn't expect data to be stationary (mean, stdev constant over time), you can train data "as is"
- `train` (mandatory) and `test` (optional) data channels
- `JSON Lines` format (`.jsonl`), `gzip` or `Parquet` file format
- each record **must** contain:
    - `start`: the starting timestamp
    - `target`: time series values
    - use `time_freq` to specify the granularity
- each record **can** contain:
    - `dynamic_feat`: dynamic features (e.g. promotion applied)
    - `cat`: categorical features

- can specify a single file or a directory with multiple files (and subdirectories)
- can train on both GPU and CPU, both single and multi-machine settings
- start with `ml.c4.2xlarge` or `ml.c4.4xlarge`, switch to GPU and multiple instances only when necessary
- inference only on CPU instances
- always include **entire time series** for training, testing, and inference
- use entire dataset as **test** set, remove last `prediction_length` points for **training**
- don't use very large values (>400) for `prediction_length`. If you want to forecast further into the future, consider aggregating your data at a lower frequency
- train on many time series and not just one when possible

Hyperparameters

- `context_length`: to control how far in the past the network can see. **However, the model will lag one year anyway**
- `prediction_length`: control how far in the future predictions can be made
- `epochs`
- `mini_batch_size`
- `learning_rate`
- `num_cells`
- `time_freq`: the **resolution of time series** (from minutes to months)

Links

- Documentation
- DeepAR: Probabilistic Forecasting with Autoregressive Recurrent Networks
- DeepAR demo on electricity dataset
- DeepAR demo on synthetic dataset
- Predicting world temperature with time series and DeepAR on Amazon SageMaker

BlazingText

Highly optimized implementations of the `Word2vec` and `text classification` algorithms.

Applications:

- `text classification` supervised:
  - **predict labels for a sentence, useful in web searches, information retrieval**
  - supervised multi-class, multi-label classification
  - can use GPU
  - for large datasets (> 2GB) use single GPU instance (`p2` or `p3`)
- `Word2vec` unsupervised:
  - **useful for downstream NLP tasks, but not is not an NLP algorithm**
  - maps words to high-quality distributed vectors - *word embedding*
  - words that are semantically similar correspond to vectors that are close together
  - can scale to large datasets easily

    - provides `skip-gram`, `batch_skipgram`, and continuous bag of words `cbow` training architectures
    - order of words doesn't matter
    - GPU training
    - for `cbow` and `skipgram` recommended a **single** `p3` instance or any single CPU/single GPU instance
    - for `batch_skipgram` can use single or multiple CPU instances

- **only works on individual words, not sentences or documents**
- expects a single preprocessed text file with space-separated tokens. Each line should contain a single sentence
- if you need to train on multiple text files, concatenate them into one file and upload the file in the respective channel
- for **supervised** mode (`text classification`):
    - one sentence per line along with labels
    - first word in the sentence: `__label__<label>`
    - tokens within the sentence - **including punctuation** - should be space separated
    - supports `validation` channel
    - supports `Pipe` mode with **Augmented Manifest Text Format**
- for **unsupervised** mode (`Word2Vec`) training:
    - only `train` channel is supported
    - one sentence per line
- not parallelizable

Hyperparameters

Word2Vec:

- **required** `mode` (`batch_skipgram`, `skipgram`, `cbow`)
- `learning_rate`
- `window_size`
- `vecto_dim`
- `negative_samples`

Text classification:

- **required** `mode` (`supervised`)
- `learning_rate`
- `epochs`
- `word_ngrams`
- `vector_dim`

Links

- Documentation
- Training Word Embeddings On AWS SageMaker Using BlazingText
- sample notebook
- BlazingText Algorithm
- BlazingText word2vec

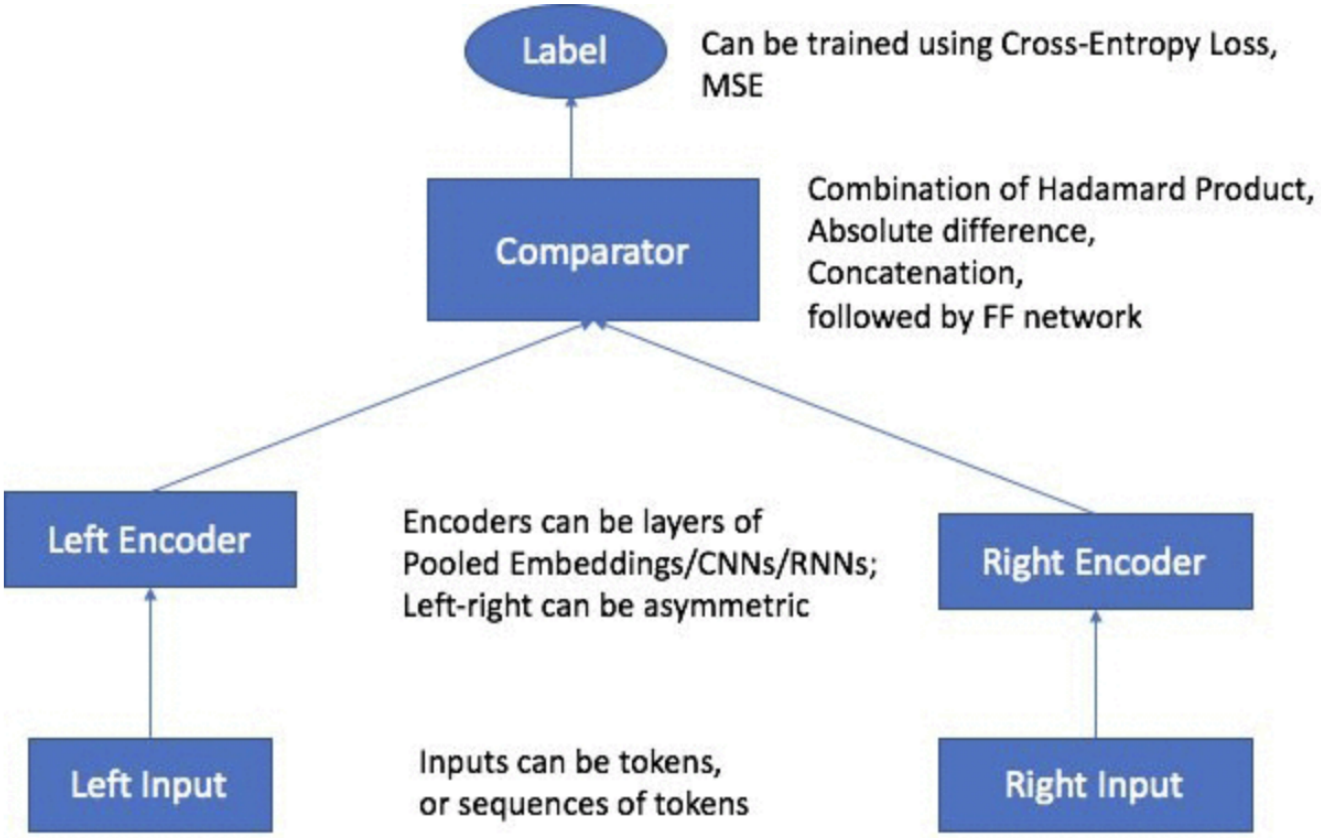
Object2Vec

A general-purpose neural embedding algorithm, similar to `Word2Vec` It can learn low-dimensional dense embeddings of high-dimensional objects. It is basically `word2vec`, generalized to handle things other than words.

The embeddings are learned in a way that preserves the semantics of the relationship between pairs of objects in the original space in the embedding space. You can use the learned embeddings to efficiently compute nearest neighbors of objects and to visualize natural clusters of related objects in low-dimensional space.

Applications:

- information retrieval
- product search
- genre predictions
- visualize clusters
- item matching
- customer profiling
- collaborative recommendation system -> training with pairs of {token, token) `{embedding for user features, embedding for item features}`
- multi-label document classification -> training with pairs of {token, sequence} `{embedding for document, embedding for label}`
- sentence embeddings -> training with pairs of {sequence, sequence} `{embedding for sentence 1, embedding for seqence 2}`



- pre-processing is required to transform the input data to the supported formats
- data must be tokenized into integers
- supports a discrete token (list of a single `integer-id`, `[10]`) or sequence of discrete tokens (`[1,4,10]`)
- requires labeled data for training -> `Object2Vec` is a **supervised** learner
- the architecture of `Object2Vec` requires the user to make the relationship between objects in each pair explicit at training time, but the relationships themselves may be obtained from natural groupings in data, and they might not require explicit human labeling
- supports a training data channel, a validation data channel, and a test data channel
- provides an auxiliary data channel for you to provide a pre-trained embedding file and a vocabulary file
- using pre-trained embedding provides a warm start to the algorithm training since it starts from an informed initial point in the input layer
- for **training**: start with `ml.m5.2xlarge`, for GPU with `ml.p2.xlarge`
- single machine only training, support for multiple GPUs
- for **inference**: `ml.p3.2xlarge`
- input for training: `JSON lines`
- inference modes:
    - To convert singleton input objects into fixed length embeddings using the corresponding encoder
    - To predict the relationship label or score between a pair of input objects
- the inference server automatically figures out which of these two modes is requested based on the input data

Hyperparameters

- `network`: encoder network (`enc1_network`, `enc2_network`): `hcnn`, `bilstm`, `pooled_embedding`

- `optimizer`
- `token_embedding_dim`
- `enc_dim`
- Early stopping tolerance and patience
- dropout, epochs, learning rate, batch size, layers, activation function, optimizer, weight decay

Links

- Documentation
- Introduction to Amazon SageMaker Object2Vec
- Movie recommender notebook
- Using Object2Vec to Encode Sentences into Fixed Length Embeddings notebook
- Using Object2Vec to learn document embeddings notebook

Object detection in SageMaker

Detects and classifies objects in images using a single deep neural network. The object is categorized into one of the classes in a specified collection with a confidence score that it belongs to the class. Its location and scale in the image are indicated by a rectangular bounding box.

- Uses CNN with Single Shot multibox Detector (SSD)
- supports two networks: VGG and ResNet
- Can be trained from scratch, or trained with models that have been pre-trained on the ImageNet dataset.
- supports RecordIO(`application/x-recordio`) and image (`png`, `jpeg`, `x-image`) content type for training
- recommended input format is Apache MXNet RecordIO
- supply a JSON file with annotation for each image
- only `x-image` for inference
- instance recommendation: GPU instances for training (`p2` and `p3`)
- can run multi-GPU and multi-instances for distributed training
- both CPU (`c5` and `m5`) and GPU (`p2` and `p3`) can be used for inference

Hyperparameters

- `mini_batch_size`
- `learning_rate`
- `optimizer`: sgd, adam, rmsprop, adadelta

Links

- Documentation
- Notebook
- Amazon SageMaker samples

Image classification

Image classification algorithm is a supervised learning algorithm that supports multi-label classification. It takes an image as input and outputs one or more labels assigned to that image.

- uses CNN ResNet
- can be trained from scratch or trained using transfer learning
- recommended input is Apache MXNet RecordIO, but can use raw `.jpg` or `.png` images
- image format requires `.lst` manifest file
- recommended to use Pipe mode
- default image size is 3-channel 224x224 (ImageNet's dataset)
- instance recommendation: GPU instances for training (`p2` and `p3`)
- can run multi-GPU and multi-instances for distributed training
- both CPU (`c5` and `m5`) and GPU (`p2` and `p3`) can be used for inference

Hyperparameters

Links

- Documentation
- End-to-End Multiclass Image Classification Example

Semantic Segmentation

- Pixel-level object classification: tags every pixel with a class label from a predefined set of classes
- useful for self-driving vehicles, medical imaging diagnostic, robot sensing
- produces a *segmentation mask*: grayscale image
- built using MXNet Gluon framework and Gluon CV toolkit
- can use FCN algorithm, Pyramid Scene Parsing (PSP), DeepLabV3
- jpg or png images for training
- incremental or training from scratch
- only GPU supported for training
- inference on both CPU and GPU
- Two distinct components:
    - backbone (or encoder): ResNet50 or ResNet110, both trained on ImageNet
    - decoder

Hyperparameters

- algorithm
- backbone

Links

- Documentation
- Example

Random Cut Forest (RCF)

- **Unsupervised** anomaly detection
- tree based ensemble method
- support for time series data
- detects:
  - unexpected spikes in time series data
  - breaks in periodicity
  - unclassifiable data points
- assigns an anomaly score to each data point
- data format is `protobuf` or `csv`. For `csv` the first column represents the anomaly label ("1")
- supports File or Pipe mode
- optional test channel
- for train: `ml.m4`, `ml.c4`, and `ml.c5`
- for inference: `ml.c5.xl`

Hyperparameters

- `feature_dim`: number of features in the data set. SageMaker RCF estimator automatically computes this
- `eval_metrics`: default: `accuracy`
- `num_trees`: increasing reduces noise
- `num_samples_per_tree`: should be chosen such that $\frac{1}{\text{num samples per tree}} \approx \frac{anomalous}{normal}$

Links

- Documentation
- An Introduction to SageMaker Random Cut Forests
- Use the built-in Amazon SageMaker Random Cut Forest algorithm for anomaly detection
- https://youtu.be/5p8B2Ikcw-k
- https://youtu.be/RyFQXQf4w4w
- https://youtu.be/12Xq9OLdQwQ

Neural topic model (NTM)

Unsupervised learning algorithm that is used to organize a corpus of documents into topics that contain word groupings based on their statistical distribution. The topics from documents that NTM learns are characterized as a *latent* representation because the topics are inferred from the observed word distributions in the corpus.

- The semantics of topics are usually inferred by examining the top ranking words they contain
- Only the number of topics, not the topics themselves, are prespecified
- deep learning algorithm
- four data channels: train is required, validation, test, auxiliary optional
- `protobuf` or `csv`
- words must be tokenized into integers
  - every document must contain a count for every word in the vocabulary in CSV
- File or Pipe mode

- one of two topic modeling algorithms in SageMaker (NTM or LDA)
- CPU or GPU
- NTM hardware is more flexible than LDA and can scale better (NTM can run on CPU and GPU, multiple instances)

Hyperparameters

- `num_topics`

Links

- Documentation
- Introduction to Basic Functionality of NTM

Latent Dirichlet Allocation (LDA)

Another topic modeling algorithm, not deep learning, **unsupervised** Attempts to describe a set of observations as a mixture of distinct categories

- most commonly used to discover a user-specified number of topics shared by documents within a text corpus
- can be used for things other than words, e.g.:
  - cluster customers based on purchases
  - harmonic analysis in music
- `protobuf` or `csv`
- LDA only supports single-instance CPU training
- Pipe mode only with `protobuf`
- lemmatization significantly increases algorithm performance and accuracy. Consider pre-processing any input text data
- LDA is a "bag-of-words" model, which means that the order of words does not matter

Terminology

- Observations are referred to as **documents**
- The feature set is referred to as **vocabulary**
- A feature is referred to as a **word**, is the occurrence count of each word
- The resulting categories are referred to as **topics**

Hyperparameters

- `num_topics`
- `alpha0`: initial guess for concentration parameter
  - smaller values generate sparse topic mixtures
  - larger values (>1.0) produce uniform mixtures

Links

- Documentation
- An Introduction to SageMaker LDA

Choosing between Latent Dirichlet Allocation (LDA) and Neural Topic Model (NTM)

Topic models are commonly used to produce topics from corpuses that (1) coherently encapsulate semantic meaning and (2) describe documents well. As such, topic models aim to minimize perplexity and maximize topic coherence. A lower perplexity score indicates better generalization performance.

- SageMaker NTM hardware is more flexible than LDA and can scale better because NTM can run on CPU and GPU and can be parallelized across multiple GPU instances, whereas LDA only supports single-instance CPU training

k-NN

A non-parametric **supervised** method for *classification* or *regression* Training with k-NN has three steps: sampling, dimension reduction, index building

- can separate non-linear data
- for dimension reductions: `sign` or `fjlt` methods
- for **training**: supports `text/csv` and `application/x-recordio-protobuf`
- for **inference** input: supports the `application/json`, `application/x-recordio-protobuf`, and `text/csv`
- train channel contains training data
- test channel emits accuracy or MSE
- supports training on CPU and GPU
- inference requests from CPUs generally have a **lower average latency** than requests from GPUs, but GPUs have **higher throughput** for larger batches
- File or Pipe mode

How the k-NN algorithm works

Step 1: Sample

- use `sample_size` parameter to specify the total number of data points to be sampled from the training dataset

Step 2: Dimension reduction

- specify the method in `dimension_reduction_type`. `sign` and `fjlt`

Step 3: Build an index

- during inference, the algorithm queries the index for the k-nearest-neighbors of a sample point

Hyperparameters

- `k`
- `sample_size`

Links

- Documentation
- K-Nearest Neighbor Covertype

k-Means

The PCA and K-means algorithms are useful in collection of data using census form.

- **Unsupervised**, divides data in `k` groups
- You define the attributes that you want the algorithm to use to determine similarity
- uses modified version of web-scale k-means clustering
- expects tabular data, where rows represent the observations that you want to cluster, and the columns represent attributes of the observations
- measured by *Euclidean distance*
- train and test channels, recommended `S3DataDistributionType=ShardedByS3Key` for training and `S3DataDistributionType=FullyReplicated` for testing
- both `File` and `Pipe` mode
- `recordIO-wrapped-protobuf` and `csv` for training (+ `json` for inference)
- both CPU and GPU (only one GPU is supported, `p*.xlarge`), but CPU is recommended
- k-means++ tries to make initial clusters far apart

How k-Means works

- **Step 1**: Determine the initial cluster centers:
  - the random approach
  - k-means++
- **Step 2**: Iterate over the training dataset and calculate cluster centers
- **Step 3**: Reduce the clusters from K to k - $K=k*x, x > 1$

Hyperparameters

- **required** `feature_dim`: number of features in the input data
- **required** `k`:
  - optimize for tightness of clusters
  - use "elbow method"

Links

- Documentation
- Analyze US census data for population segmentation using Amazon SageMaker

PCA

- **Unsupervised** dimensionality reduction
- reduced dimensions are called *components*
- two modes:
  - **regular**: for datasets with sparse data and a moderate number of observations and features
  - **randomized**: for dataset with both a large number of observations and features, uses an approximation algorithm
- covariance matrix is created, then singular value decomposition (SVD)
- uses tabular data
- works only for numeric data

- data needs to be normalized - features with similar scale
- both `protobuf` and `csv` for training
- `csv`, `json`, `protobuf` for inference
- File or Pipe mode
- both CPU and GPU

Hyperparameters

Links

- Documentation
- An Introduction to PCA with MNIST

Factorization machines

A a general-purpose **supervised** learning algorithm that you can use for both **classification** and **regression** tasks.

- extension of a linear model that is designed to capture interactions between features with high dimensional sparse datasets economically
- good choice for tasks dealing with high dimensional sparse datasets, such as *click prediction* and *item recommendation*
- usually used in the context of recommender systems
- finds factors we can use to predict a classification (click on not? purchase or not?) or value (predicted rating?) given a matrix representing some pairs of things (e.g. users & items)
- considers **only pair-wise** (2nd order) interactions between features
- linear complexity for calculating model parameters
- only `protobuf` with `Float32` tensors for training, `csv` is not practical with sparse data
- both File and Pipe mode
- for inference `json` and `protobuf`
- can be trained across distributed instances
- CPU is recommended for sparse data
- GPU must be better for dense data

Hyperparameters

- initialization methods for bias, factors, and linear terms: uniform, normal, constant

Links

- Documentation
- Sample notebook

IP-insights

- **Unsupervised** learning algorithm that learns the usage patterns for IPv4 addresses
- uses NN to learn latent vector representation of entities and IP addresses
- designed to capture associations between IPv4 addresses and various entities, such as user IDs or account numbers

- ingests historical data as `(entity, IPv4 Address)` pairs and learns the IP usage patterns of each entity
- When queried with an `(entity, IPv4 Address)` event, a model returns a score that infers how anomalous the pattern of the event is
- can be used:
    - identify logins from anomalous IPs
- can also learn vector representations of IP addresses, known as *embeddings*
- supports training and validation data channels. Validation channel is used to compute AUC score
- training/validation data must be in `csv` format, `(entity, IP)`
- File mode only
- for inference: `csv`, `json`, `jsonlines` are supported
- can run on both GPU and CPU
- for training, GPU is recommended, but for certain workloads with large training datasets, distributed CPU instances might reduce training costs
- for inference, CPU is recommended
- supports all available GPUs

If you switch from a single GPU to multiple GPUs, the mini_batch_size is divided equally into the number of GPUs used. You may want to increase the value of the mini_batch_size to compensate for this.

Hyperparameters

- `num_entity_vectors`
    - hash size
    - set to 2x number of unique entity IDs
- `vector_dim`
    - size of embedding vectors
    - scales model size
    - too large results in overfitting

Links

- Documentation
- Sample notebook

Reinforcement learning in SageMaker

Q-learning:

- You have:
    - a set of environmental states s
    - a set of possible actions in those states a
    - a value of each state/action Q
- Start with Q values of 0
- explore the space
- a bad thing happen after a given state/action -> reduce its Q
- a good thing (reward) happen -> increase its Q

- you can look ahead more than one step by using a discount factor (s previous state, s' current state): $Q(s,a) = discount*(reward(s,a) + max(Q(s')) - Q(s,a))$

exploration problem:

- how do we efficiently explore all of the possible states?

Markov Decision Process (MDP)

- discrete time stochastic control process
- States are s and s'
- state transition functions: $P_{a}(s,s')$
- Q values as reward function: $R_{a}(s,s')$

RF in SageMaker:

- uses a deep learning framework with Tensorflow and MXNet
- supports Intel Coach and Ray Rllib toolkits
- custom, open-source, or commercial environments supported

Automatic Model Tuning

- don't optimize too many hyperparameters at once
- limit your ranges to as small a range as possible
- use logarithmic scales when appropriate
- don't run too many training jobs concurrently: this limits how well the process can learn as it goes
- best practices for hyperparameter tuning

Apache Spark with SageMaker

- pre-process data as normal with Spark - generate DataFrames
- use `sagemaker-spark` library
- train `SageMakerEstimator` (KMeans, PCA, XGBoost)
- get `SageMakerModel` from `fit` on `SageMakerEstimator`
- call `transform` on `SageMakerModel`

SageMaker Studio and Experiments

SageMaker Debugger

Saves internal model state at periodical intervals

- gradients/tensors over time for training
- define rules for detecting unwanted conditions while training
- a debug job is run for each rule configured
- logs & fires a CloudWatch event when the rul is hit

SageMaker Autopilot

...

SageMaker Model Monitor

- get alerts on quality deviations on your deployed models (via CW)
- visualize data drift
- detect anomalies & outliers
- detect new features
- no code
- integrates with SageMaker Clarify (detects potential bias)
- monitoring types:
    - data quality
    - model quality
    - bias drift
    - feature attribution drift

SageMaker logging and monitoring options + CloudTrail

- Statistics are kept for 15 months, CloudWatch console limits the search to metrics for the last two weeks
- monitoring metrics are available on CW at a 1-min frequency
- the default IAM role has the required permissions to write logs
- CloudTrail does not monitor calls to `InvokeEndpoint` - **CloudTrail captures all API calls for SageMaker, with the exception of `InvokeEndpoint`**
- CloudTrail keeps this record for a period of 90 days

New features in 2021

- JumpStart
- Data Wrangler
- Feature Store
- Edge Manager

`CreateTrainingJob` API required parameters:

- `AlgorithmSpecification`
- `OutputDataConfig`
- `ResourceConfig` - ML compute instance type, count, volume KMS key, volume size
- `RoleArn`
- `StoppingCondition` - `MaxRuntimeInSeconds` (default 1 day, max is 28 days) and `MaxWaitTimeInSeconds`
- `TrainingJobName`

Bias-variance trade-off https://mlu.corp.amazon.com/explain/bias-variance

Monitor model performance

https://aws.amazon.com/blogs/machine-learning/use-amazon-cloudwatch-custom-metrics-for-real-time-monitoring-of-amazon-sagemaker-model-performance/

Naive Bayesian classifier

https://www.analyticsvidhya.com/blog/2017/09/naive-bayes-explained/

$$ P(class|data)=\frac{P(data|class)*P(class)}{P(data)} $$

Evaluating ML models

docs

- can use console to split the data (if single file or directory)
- default path in Create ML model wizard splits 70/30 sequentially, not random
- can randomize using **Custom** 70/30 random split
- to specify custom split ratios in `Create Datasource` API
- for big data sets - can reserve some data (20%) for test (validation)
- for small data sets - use **Cross-Validation**

Binary model insights

- select cut-off/threshold, default 0.5
- AUC, **independent** of cut-off

False positive rate $$ FPR = \frac{FP}{FP+TN} $$

Multiclass model insights

F1 $$ F1 = \frac{2TP}{2TP+FP+FN} $$

Macro average F1 score $$ F1 = \frac{1}{K}\sum^K_{k=1}{F1_k} $$

Regression model insights

RMSE $$ RMSE=\sqrt{\frac{1}{N}\sum_{i=1}^N{(y_i-\hat{y_i})^2}} $$

- Common practice to review the **residuals** for regression problems
- Residuals represent the portion of the target that the model is unable to predict
- Positive residuals -> model underestimating
- Negative residuals -> model overestimating
- The histogram of the residuals on the evaluation data when distributed in a bell shape and centered at zero indicates that the model makes mistakes in a random manner and does not systematically over or under predict any particular range of target values
- If the residuals do not form a zero-centered bell shape, there is some structure in the model's prediction error

Cross-validation

- Use cross-validation to detect overfitting
- k-fold cross-validation

Evaluation alerts If any of the validation criteria are not met by the evaluation, the Amazon ML console alerts you by displaying the validation criterion that has been violated, as follows:

- Evaluation of ML model is done on held-out data
- Sufficient data was used for the evaluation of the predictive model (if evaluation data < 10% of training data)

- Schema matched
- All records from evaluation files were used for predictive model performance evaluation
- Distribution of target variable: target is distributed similarly in both training and evaluation data sets
    - **if this alert triggers**: try using the random split strategy to split the data into training and evaluation datasources

Model Fit: Underfitting vs. Overfitting

docs

if **underfitting**:

- Add new domain-specific features and more feature Cartesian products, and change the types of feature processing used (e.g., increasing n-grams size)
- Decrease the amount of regularization used

if **overfitting**:

- Feature selection: consider using fewer feature combinations, decrease n-grams size, and decrease the number of numeric attribute bins.
- Increase the amount of regularization used

SageMaker integration in Spark

- Install SageMaker Spark library in the Spark environment
- Use the appropriate estimator from the SageMaker Spark Library to train a model
- Use the SageMaker `model.transform` method to get inferences from the model hosted in SageMaker

SageMaker endpoint scaling

- docs
- Load test and optimize an Amazon SageMaker endpoint using automatic scaling

Perform load tests to choose an automatic scaling configuration that works the way you want.

- Assumption: automatic scaling policy uses the predefined target metric `SageMakerVariantInvocationsPerInstance`
- Perform load testing to find the peak `InvocationsPerInstance`:
    1. set up an endpoint with your model using a single instance
    2. use a load testing tool to generate an increasing number of parallel requests - determine the peak request-per-second (RPS) you model's production variant can handle

`SageMakerVariantInvocationsPerInstance = (MAX_RPS * SAFETY_FACTOR) * 60` -> per **minute**

Modeling: High-level ML services

Amazon Comprehend

- NLP and text analytics
- extract key phrases, entities, sentiments, languages, syntax, topics, document classification

- Features:
    - comprehend insights:
        - entities (people, places, locations)
        - key phrases
        - PII
        - language
        - sentiment (positive, neutral, negative, or mixed)
        - targeted sentiment - associated with specific entities in a document
        - syntax - parses each word in a document and determines the part of speech
    - comprehend custom:
        - custom classification, multi-class and multi-label
        - custom entity recognition
    - document clustering (topic modeling)

Amazon Translate

- use deep learning for translation
- supports custom terminology

Amazon Transcribe

- speech to text
- streaming audio supported
- speaker identification (number of speakers)
- channel identification
- custom vocabularies

Amazon Polly

- text to speech
- Lexicons, e.g. for handling acronyms
- SSML (speech synthesis markup language)
- Speech marks

Amazon Rekognition

- object and scene detection
- image moderation
- facial analysis
- celebrity recognition
- text in images
- video analysis
- video must come from KVS (Kinesis Video Streams), favor resolution over framerate, 5-30 FPS, H.265 encoded
- images come from S3
- can use with Lambda to trigger image analysis upon upload
- recognition custom labels - train with a small set of labeled images

Amazon Forecast

- Time series forecast
- works on any time series

Amazon Lex

- Natural-language chatbot engine
- built around intents
- utterances invoke intents
- lambda functions are invoked to fulfill the intent
- slots specify extra info needed by the intent

Amazon Personalize

- fully-managed recommender engine
- API access:
    - feed in data (purchases, ratings, impressions, card adds, catalog, user demographics) via S3 or API
    - you provide an explicit schema in `Avro` format
    - Javascript or SDK
    - `GetRecommendations`: recommended products, content, similar items
    - `GetPersonalizedRanking`: rank a list of items provided

Features:

- real-time or batch recommendations
- cold start recommendations (for new users/new items)
- contextual recommendations (device type, time, etc)
- similar items
- unstructured text input

Datasets:

- users including demographics
- items
- interactions

Recipes:

- user personalization
- personalized rankings
- related items

Solutions:

- trains the model
- optimizes for relevance as well as your additional objectives
- HPO

Campaigns:

- Deploys your "solution versions"

- deploys capacity for inference

HPO

User personalization, personalized ranking:

- `hidden_dimension`
- `bptt` (back propagation through time - RNN)
- `recency_mask` (weights recent events)
- `min/max_user_history_length_percentile` (filter out robots)
- `exploration_weight` (0-1), controls relevance
- `exploration_item_age_cut_off`, how far back in time you go

Similar-items:

- item_id_hidden_dimension
- item_metadata_hidden_dimension

Maintain relevance

- keep your dataset current - incremental data import
- use `PutEvents` to feed in real-time user behavior
- retrain the model:
    - create a *new solution version*
    - updates every 2 hours by default
    - should do a full retrain (`trainingMode=FULL`) weekly

Security

- data not shared across accounts
- data may be encrypted with KMS
- access control via IAM
- data in S3 must have appropriate bucket policy for Amazon Personalize
- monitoring/logging

Pricing

- data ingestion per GB
- training per hour
- inference per TPS-hour
- batch recommendation per user or per item

Amazon Textract

- OCR with forms, fields, tables support

AWS DeepRacer

- RF powered by race car

DeepLens

- deep learning-enabled video camera
- integrated with Rekognition, SageMaker

Amazon Lookout

- for equipment
- for metrics
- for vision

detects abnormalities from sensor data automatically to detect equipment issues

Amazon Monitron

- end-to-end system for monitoring industrial equipment & predictive maintenance

TorchServe

- model serving framework for PyTorch
- part of the PyTorch open source project from Facebook

AWS Neuron

- SDK for ML inference specifically on AWS Inferentia chips
- EC2 Inf1 instance type
- Integrated with SM/DLC/AMI, Tensorflow/PyTorch/MXNet

AWS Panorama

- computer vision at the edge
- brings CV to your existing IP cameras

AWS DeepComposer

- AI powered keyboard
- composes a melody into an entire song

Amazon Fraud Detector

- upload you own historical fraud data
- builds custom models from template you choose
- exposes API for your online application

Amazon CodeGuru

- automated code review
- finds lines of code that hurt performance
- resource leaks, race conditions
- offers specific recommendations
- Java and Python

Contact Lens for Amazon Connnect

- for support call centers
- ingest audio data from recorded calls
- allows search on calls/chats
- sentiment analysis
- find utterances that correlate with successful calls
- categorize calls automatically
- measures talk speed and interruptions
- theme detection: discovers emerging issues

Amazon Kendra

- enterprise search with natural language

Amazon Augmented AI (A2I)

- human review of ML predictions
- builds workflows for reviewing low-confidence predictions
- integrated into Textract and Rekognition, SM

Section 8: ML implementation and operations

SageMaker and Docker

- all models in SM are hosted in Docker containers:
    - pre-built deep learning
    - pre-built scikit-learn and Spark ML
    - pre-built Tensorflow, MXNet, Chainer, PyTorch
        - distributed training via Horovod or Parameter Servers
    - own training and inference code or pre-built image extension

consider environment variables

- Documentation

Sample docker file:

```
# Downloads the TensorFlow Docker base image.
# You can replace this with any Docker base image you want to bring to
build containers, as well as with AWS pre-built container base images
FROM tensorflow/tensorflow:2.2.0rc2-gpu-py3-jupyter

# Install sagemaker-training toolkit that contains the common
functionality necessary to create a container compatible with SageMaker
and the Python SDK.
RUN pip3 install sagemaker-training

# Copies the training code inside the container
# The script must be located in this folder
```

```
COPY train.py /opt/ml/code/train.py

# Defines train.py as script entrypoint
ENV SAGEMAKER_PROGRAM train.py
```

❗ `SAGEMAKER_PROGRAM` is the only environmental variable that you must specify when you build your own container

Horovod

- a distributed deep learning training framework for TensorFlow, Keras, PyTorch, and Apache MXNet
- the primary motivation for this project is to make it easy to take a single-GPU training script and successfully scale it to train across many GPUs in parallel
- Amazon SageMaker provides for TensorFlow model hosting and training, including fully managed distributed training with Horovod and parameter servers

Production variants

- test out multiple models on live traffic
- variant weights - how to distribute traffic among them
- lets you do A/B tests and validate performance in real-world settings

SageMaker on the Edge

SageMaker Neo

- use cases:
  - no connectivity to the cloud
  - latency-sensitive cases
  - processing sensor data in autonomous vehicles
- compile models to edge devices: ARM, Intel, Nvidia, Candence, Qualcommm, Xilinx embedded
- optimizes code for specific devices
- supports MXNet, TensorFlow, PyTorch, XGBoost models
- consists of a compiler and a runtime
- up to 10x reduction in framework size
- compiles model and framework into a single executable that can be deployed to an edge device

Neo + AWS IoT Greengrass

- addressed a challenge **how to deploy** a model on your edge device
- IoT Greengrass extends AWS to edge devices
- with Greengrass, you can run Lambda functions, docker containers, execute predictions on machine learning models in your edge device even when not connected to the internet
- Neo-compiled models can be deployed to an HTTPS endpoint
  - hosted on c5, m5, m4, p3, p2 instances
  - must be same instance type used for compilation
- You can deploy to IoT Greengrass
  - how to get the model to an actual edge device

    - place your Neo compiled executable on S3
    - install DLR - a compact runtime for ML models, point model to S3 location
    - Greengrass runtime then downloads the executable and hosts the model in your computer, and you can use it to generate inference locally
    - uses Lambda inference applications

SageMaker security

Network isolation, the following SageMaker containers do not support isolation:

- Chainer
- PyTorch
- Scikit-learn
- Reinforcement learning

SageMaker resource management

- Use checkpoints to S3 so training can resume
- can increase training time (need wait for spot instances)

Elastic inference

- accelerates DL inference
    - a fraction of cost of using a GPU instance for inference
- EL accelerators may be added alongside a CPU instance:
    - ml.eia1.medium/large/xlarge
- EL accelerators may be applied to notebooks
- works with TF and MXNet pre-built containers
- works with image classification and object detection built-in algorithms
- you must create an AWS PrivateLink endpoint service

Automatic scaling

- works with CloudWatch
- you setup a scaling policy to define target metrics, min/max capacity, cool-down periods

SageMaker inference pipelines

- linear sequence of 2-15 containers
- any combination of pre-trained built-in algorithms or your own algorithms in Docker containers
- combine pre-processing, prediction, post-processing
- Spark ML and scikit-learn containers
- can handle both real-time inference and batch transforms
- invocations as a sequence of HTTP requests

SageMaker inference containers

documentation custom inference containers requirements:

- Responding to inference requests:

    - web server listening on 8080
    - accept `POST` request to `/invocations` endpoint
    - must accept socket connection requests within `250ms`
    - must respond to requests within 60s
- Responding to health check (ping):
    - `GET` requests to `/ping` endpoint
    - simplest requirement is return `HTTP 200` and empty body
    - respond on `/ping` within `2s`

Wrapping up

More prep resources

- practice exam

LinuxAcademy course by **Mike Chambers**

https://linuxacademy.com/cp/modules/view/id/340

- Training Data
- Testing Data

Overfit and failing to generalize

1. Data
2. Algorithm
3. Train
4. Model

Deep Learning

Artificial Neuron

- inputs
- outputs
- activation function

Mesh network of neurons

Input layer Hidden layers Output layer - one neuron for each outcome

Machine Learning Lifecycle

Collect Data Process Data

- Features
- Label (in case of labeled data)
- Feature engineering

- Feature reduction
- Encoding
- Formatting

Split Data

- Training (80%)
- Validation (optional)
- Testing (20%)

Train

- algorithm "sees" and directly influenced by the **training** data

Test

- algorithm "does not see" **testing** data

Deploy

- DevOps
- Infrastructure

Infer -> Predictions -> Improve cycle

- real-world unlabeled data
- take statistics etc -> repeat the life cycle

Different Types of Machine Learning

Supervised

- Labeled data
- Numberic data
- Classified data

Unsupervised

- looking for patterns in the data

Reinforcement Learning

- Action-Reward
- often used in robotics and automatics
- e.g. Deep Racer

Optimization

Gradient Descent

- Step size sets the learning rate

used to optimize:

- Linear Regression
- Logistical Regression
- Support Vector Machines

Regularization

Regularization through regression change sensitivity of the model in different dimensions

- L1
- L2

apply when model is overfit

Hyperparameters

external parameters to set on model when starting training

- **Learning rate** - size of step taken during gradient descent optimization (0,1)
- **Batch size** - number of samples used to train at any one time (all, one, or some)
- **Epochs** - number of times the algorithm will process the entire training data (contains one or more batches)

Cross-validation

Split data:

- training (seen by the training process)
- validation (indirectly influences the model, **not** seen by training process)
- testing

K-Fold Cross-Validation Split training data in K parts and use K-1 for training and 1 for validation - repeat K times

Data

Feature Selection and Engineering

Selecting relevant data

- removing irrelevant data/features
- looking at correlation and variance in the data - drop features with very low correlation to the labeled data or with very low variance
- correlation matrix show quatification of the **linear** relationships among variables

Gaps and Anomalies Engineer new feature (e.g. take a function from multiple features)

See:

- PCA - Principal Component Analysis

- Label Encoding
- One Hot Encoding

Principal Component Analysis

Dimension reduction Determine the most important components of the data

PCA is an unsupervised ML model PCA is often used as a data preprocessing step There can be as many PC's as features or values PC1 and PC2 can be used to plot a 2D graph to show groups of features

Label and One Hot Encoding

Label encoding - replace a name (in the category) with a number *Problem with label encoding* - implying a hiearchy or ranking One Hot Encoding - introduce a feature for each value in the category

Use *label encoding* to replace string values Use *one hot encoding* for categorical features

Missing and Unbalanced Data

Options for dealing with missing data:

- Impute data - e.g. take a mean of all data values for the feature
- Remove data or remove the whole feature

Unbalanced:

- Source more data
- Oversample minority data
- Synthesize data (use domain knowledge)
- Try different types of algorithm

Splitting and Randomization

Training Validation Testing

Important to understand how the data is collected

- Data collected over time
- Batches of data (class sorted)

Always randomize the data Even you're unaware of data clumping Some algorithms will shuffle data during training, but not between the training and test data sets

RecordIO format

Pipe mode streams data Faster training start times and better throughput Most SageMaker algorithms work best with RecordIO

- Streams data directly from S3
- Training instances don't need a local disk copy of data

Machine Learning Algorithms

Logistical Regression

Supervised Inference: Binary *Yes* or *No*

Use Cases:

- credit risk
- medical condition
- if person will take an action

Sigmoid function

Linear Regression

Supervised Inference: numeric 1,2,3...

Use Cases:

- Financial forecasting
- Marketing effectiveness
- Risk evaluation

Support Vector Machines

Supervised Inference: Classification

Use cases:

- customer classification
- Genomic identification

use support vectors to define margins and draw a hyperplane between margins

Decision Trees

Supervised Inference: Binary, Numeric, Classification

Tree root node - the feature correlating the most with the label Continue with features on the internal nodes/leaf nodes in the order of correlation rank

Random Forests

Supervised Inference: Binary, Numeric, Classification

Collection of decision trees

Takes the random subset of features available and build multiple decision trees For the inference it runs all the trees and averages the result

K-Means

Unsupervised Inference: classification

Use cases:

- data exploration
- customer categorization

Elbow Plot: total within-cluster sum (WSS) as a function of k

K-Nearest Neighbour

Supervised Inference: classification

k - number of Neighbours taken into account

- make k large enough to reduce the influence of outliers
- make k small enough that classes with a small sample size don't lose influence

Use case:

- recommendation engine
- similar articles and objects

Latent Dirichlet Allocation (LDA) Algorithm

Unsupervised Inference: classification

Use cases:

- topic discovery/analysis
- sentiment analysis
- automated document tagging

Document<->Topic<->Word

- remove "stop words"
- apply "stemming"
- tokenize
- chose the number of topics (k)
- randomly assign topics to each word
- count the words by topic
- reassign words to topics iteratively

Deep Learning Algorithms

Neural Networks Terminology

supervised artificial neurons

Input Layer - image split on all individual neurons **weights**

sum(value on neuron * weight) + **bias** -> activation function

Activation Functions:

- Sigmoid (squashing function - between 0 and 1)
- ReLU (if value <= 0 -> 0, if value > 0 -> value)
- Tanh (between 0 and 1)

combination of bias and weights - training

forward propagation loss function -> optimization e.g. gradient decent, learning rate **back propagation**

epoch - repeat forward propagation + back propagation

Convolutional Neural Networks (CNN)

Supervised Inference: image classification

Convolutional layers Pre-trained edge detection (transfer learning)

Recurrent Neural Networks (RNN)

Supervised Inference:

- stock prediction
- time series data
- voice recognition (seq to seq)

the algorithm has memory built in has ability to remember *a bit*

LSTMM (long short-term memory) - remember *a lot*

Model Performance and Optimization

Confusion Matrix

True Positives	**False Positives**
False Negatives	True Negatives

Size of the matrix NxN - where N - number of categories

Sensitivity and Specificity

Sensitivity True Positive Rate (TPR) Recall or hit rate - what % of the actual positives does the classifier get right

True Positives / (True Positives + False Negatives)

Higher sensitivity decreases *false negatives*

Specificity True Negative Rate (TNR)

True Negatives / (True Negatives + False Positives)

Higher specificity decreases *false positives*

Accuracy and Precison

Accuracy (True Positives + True Negatives) / all predictions

a model that produces completely correct prediction has an accuracy = 1.0

Precision the proportion of *actual positives* that were correctly identified

True Positives / True Positives + False Positives

a model that produces no false positives has a precision = 1.0

ROC and AUC

ROC - Receiver Operating Characteristics AUC - Area Under the Curve

x axis - False Positive Rate y axis - True Positive Rate

False Positive Rate (FPR) = False Positive / (True Negatives + False Positives) false alarm rate - what % of the actual negatives does the classifier get wrong (i.e. predict to be positive)

Gini Impurity

$Gini impurity = 1 - p(label_1)^2 - p(label_2)^2$ p - probability of a label take weighted Gini impurity over all branches

The feature with the **lowest** weighted Gini impurity is the best choice for the root node (best separates labels)

F1 Score

$F1 = 2*\frac{Recall * Precision}{Recall + Precision}$

Recall (sensitivity) = True Positives / (True Positives + False Negatives) Precision = True Positives / True Positives + False Positives

F1 Score takes more into account and can be a better measure for models if we have an uneven class distribution.

Machine Learning Tools and Frameworks

Jupyter Notebooks

- wiki for code
- combines executable code with documentation
- designed to be shared

User/Browser <-> Jupyter Notebook Server <-> Kernel Notebook files

SageMaker

- fully-managed notebook instances
- Multiple kernel options
- used by AWS to demo SageMaker algorithms

ML and DL Frameworks

set of APIs, libraries, compilers etc implementation of algorithms

- TensorFlow (google)
- Keras (goes together with TensorFlow)
- mxnet (AWS)
- Gluon
- PyTorch
- Scikit learn (great place for experimenting)

Languages:

- Python
- R
- Go

TensorFlow

PyTorch

MXNet

Scikit-learn
