Practical Applied ML at & with Google tools

For PHY466: Introduction to Applied Deep Learning

Mathieu Guillame-Bert
Richard Stotz

Outline

1. What is an ML model
2. Advice for applied ML development
3. Flexible & extensible Deep Learning with the JAX library
4. Gradient Boosted Trees with the YDF library
5. Tutorials

What is an ML model

...in practice

What we care about [informally]

$$\operatorname*{argmax}_{f} \mathbb{E}_{(x,y)\in\text{world}} \left[\text{metric}(y, f(x))\right]$$

(1) Optimize over the entire world.

(2) The objective.

Can be anything e.g., accuracy, AUC, accuracy under cost constraints, revenue, lives.

*Informally, we want a predictive function that **always** gives good results.*

What we care about

$$\underset{f}{\operatorname{argmax}}\, \mathbb{E}_{(x,y)\in \text{world}} \left[\text{metric}(y, f(x))\right]$$

What we do in practice

$$\underset{\theta}{\operatorname{argmin}}\, \mathbb{E}_{(x,y)\in \text{train}} \left[\text{loss}(y, \boxed{f}(\theta, x))\right]$$

(4) Simpler, but still generally intractable. Solved with heuristics.

(3) Only on the observations we have.

(2) Should be cheap to compute. Possibly constrained by the argmin solver.

(1) What type of model? (linear model, forests, neural net, ...)

*Informally, we **select the function** from a **family of functions** that **best fits our data + bias**, and **hope it generalizes** to unseen data. If not, we change the function family or the bias.*

Popular families of predictive functions

- Linear model (LM)
- Generalized additive model (GAM)
- Decision forest (DF)
- Multi layer perceptron (MLP)
- Convolutional neural net (CNN).
- Transformer-based model (TRA)
- Kolmogorov-Arnold Network (KAN)

$$f_{\mathrm{lm}}(\theta, x) = \theta x$$

$$f_{\mathrm{gam}}(\theta, x) = \sum_i g_i(\theta_i, x_i)$$

$$f_{\mathrm{df}}(\{\mathrm{l}, \mathrm{e}\}, x) = \sum_i \mathrm{l}_i \left[\mathrm{e}_i(x))\right]$$

$$f_{\mathrm{mlp}}(\theta, x) = (\theta_1 \circ \sigma \circ \theta_2 \circ \cdots \circ \sigma \circ \theta_n)(x)$$

$$f_{\mathrm{kan}}(\theta, x) = (\mathrm{gam}(\theta_1, .) \circ \cdots \circ \mathrm{gam}(\theta_n, .))(x)$$

Impact of f on training

- How **fast** is the model training?
  - A DF trains in seconds.
  - A TRA can takes hours, days, months or even years
- How much **data** is required for good results?
  - Powerful models generally take longer to converge
- What kind of pattern can the **model express**?
  - A LM can only express linear relations
  - A GAM cannot express crosses between features
  - A MLP cannot learn a hidden representation and apply it on other part of the data.
  - *Note: MLP and DF are universal approximators (they can learn any function; though not always efficiently).*

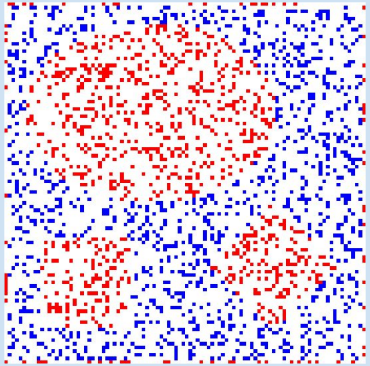
Impact of f on inference

- How **fast** is the model inference?
  - A DF makes predictions in x µs on CPU.
  - A TRA can takes xx ms on TPUs
- Can f be **interpreted / debugged** easily?
- How well can $f + \mathrm{argmax}$ **generalize** to unseen data?
  - A hashtable can record all the observations easily, but it does not generalize well.
  - For different reasons, fractals don't generalize well.

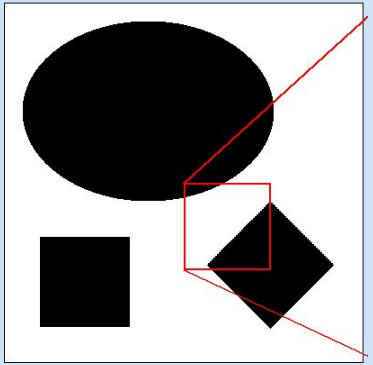
Training algorithm biases

- For each model + learning algorithm, some patterns are easier to learn than others.
- Learning is biased towards “easy” patterns.
- Below: How different learning algorithms learn different patterns (right) on the same training data.

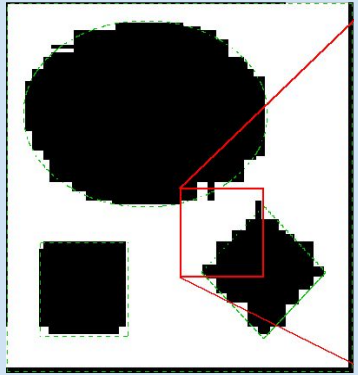
Training data



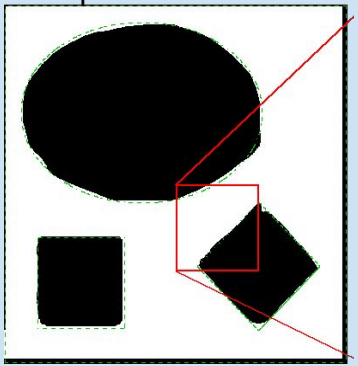
Ground truth



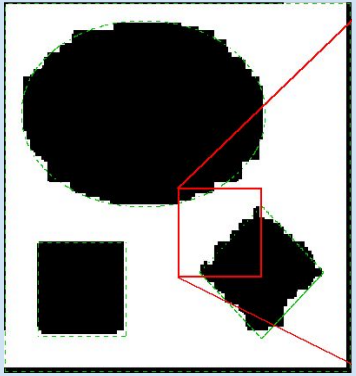
Decision Tree



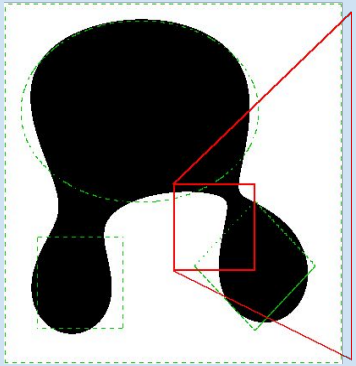
Oblique Random Forest



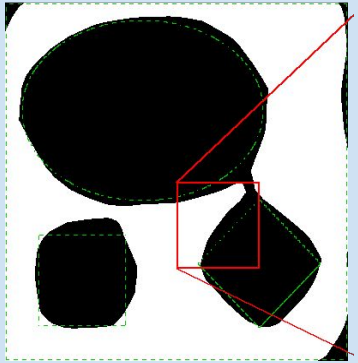
Random Forest



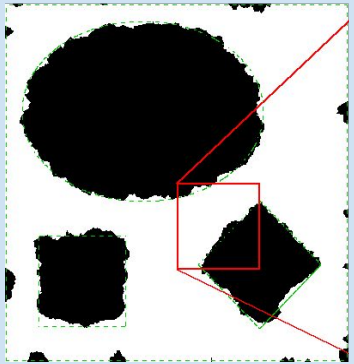
Support Vector Machine



Neural Network



k-Nearest Neighbor



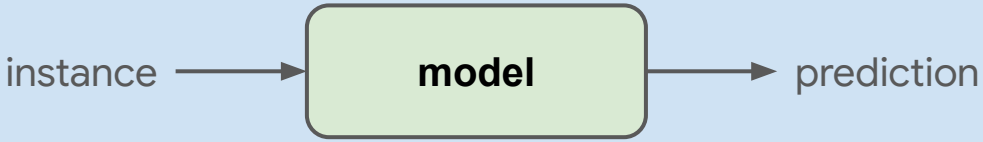
How to solve the argmin?

- Solving the argmax is generally **intractable** (i.e., it would take longer than the age of the universe to solve even a trivial problem). Instead we use **heuristics**.
- Example of heuristics:
  - Gradient Descent for LM.
  - Backpropagation for MLP, CNN and TRA.
  - Greedy divide and conquer for DF.
  - Genetic algorithms [for all types of models]
- Often, we **combine** two (or more) heuristics / algorithms for different parts of the model.
- Example of learning with 5 different learning heuristics:
  - [Training] Backpropagation to learn the kernel's weights of a CNN.
  - [Tuning] Gaussian Process Bandits to learn the number and size of kernels of a CNN.
  - [Calibration] Pool Adjacent Violators to calibrate the model predictions.
  - [Feature selection] Backward Selection to remove some of the features
  - [Human] ML expertise is a form of learning heuristic.

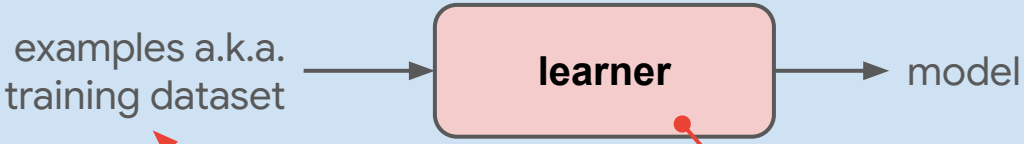
can it overfit?

ML toolbox as operators

Inference



Training



also contains the "validation" examples

$$\underset{\theta}{\operatorname{argmin}}\, \mathbb{E}_{(x,y)\in\text{train}}\left[\text{loss}(y, f_\theta(x))\right]$$

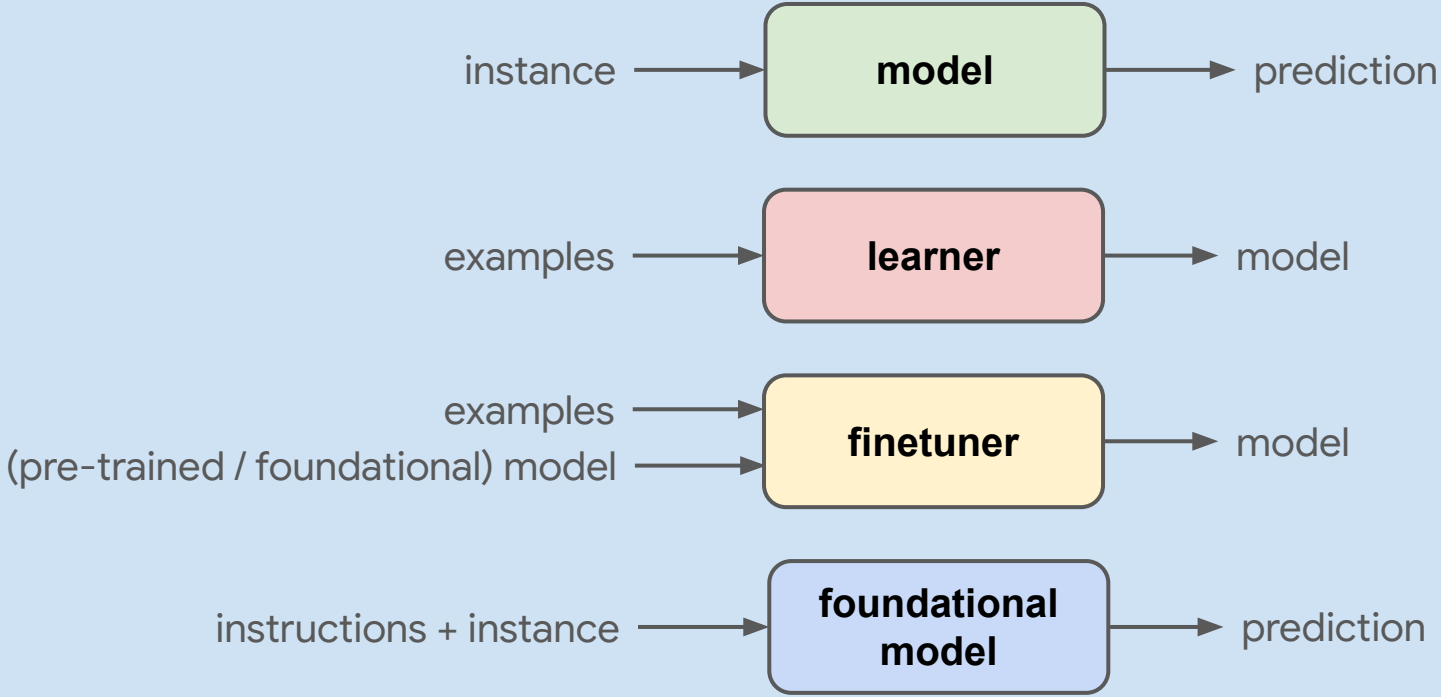
Remarks

This is a functional view. Models are not modified by inference, and learners are not modified by training (see more on functional programming in JAX section).

A learner can call another learner (i.e. meta-learner) such as hyper-parameter tuners.

A learner can generate different model types (e.g., train a NN and a DF and return the best one).

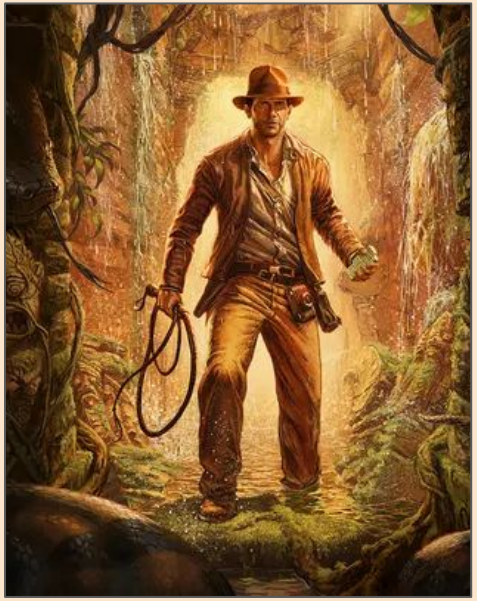
ML toolbox as operators



Advice for applied ML development

Rules are made to be broken … but they are a good starting point.

Applied ML is exploration + engineering



- What you needed
  - Knowing your tools
    - Familiarity with different ML approaches and tools.
    - Understanding their practical differences
  - Knowing how to evaluate solutions
    - Train-test and cross-validation protocols
  - Knowing how to spot danger & issues
    - Label-leakage
    - Experimenter bias
    - Distribution shift
    - Model drift
  - Patience & courage; this is an iterative process
- You don't need to understand the algorithms to use them.

Three ways to get an ML model (informally)

- **Automated Machine Learning Tools**
  - **Examples:** Azure AutoML, Google Vertex AI, Simple ML
  - **Pros:** Easy to use, often code-free, get a model quickly without deep expertise.
  - **Cons:** Limited in the types of problems that can be solved, can take a long time to run.
- **Foundational Models**
  - **Examples:** ChatGPT, Gemini, LLaMA, Gemma, TimesFM
  - **Pros:** Immediate results and fast iteration, often surprisingly good results.
  - **Cons:** Expensive & slow inference, also limited in the types of problems that can be solved.
- **Model Engineering**
  - **Examples:** Designing neural network architectures, fine-tuning models, feature engineering.
  - **Pros:** Most flexibility and control, allow super-tailored solutions, allow composition.
  - **Cons:** Need expertise and time to optimize models.
  - **Note:** Some methods are much simpler than others.

Advices for applied ML development (1/4)

1. **Advice #1:** Look at your data
   a. Look at the raw data.
   b. Make plots, histograms, contingency tables, etc.
   c. Data is often not what you think it is.
   d. Find bugs in data acquisition.
   e. You will be more efficient during the model engineering phase.
   f. [Timeseries sampling story]
2. **Advice #2:** Know your objective
   a. What are the metrics you really care about?
   b. What are satisfying values for those metrics (success condition)?
      i. Research: What is the current SOTA?
      ii. Production: What is the status-co?
   c. How expensive are those metrics to compute?
   d. Can you define cheaper proxies for those metrics?
   e. How is the model going to be used?

Advices for applied ML development (2/4)

1. **Advice #3:** Know your data
   a. How much data do you have?
   b. Can you get more (similar or related) data?
   c. How trustworthy are the labels?
   d. Is the data biased or representative of serving?
2. **Advice #4:** Start simple
   a. Start with a linear model, a decision forest, or even an histogram.
   b. Help you understand your data and spot more problems
   c. Gives you a baseline for more complex approaches later
   d. Get early results
3. **Advice #5:** Create the simplest possible e2e pipeline before adding any complexity
   a. Develop and test all the steps (e.g., data acquisition, data cleaning & augmentation, model training, model evaluation) before improving them.
   b. Easier to debug & iterate.
   c. Easier to focus on what matters.
   d. Bonus: Makes collaboration possible

Advices for applied ML development (3/4)

1. **Advice #6:** Come back to your data with the help of the model
   a. What examples are hard for the model?
   b. Can guide for feature / model engineering
   c. Can help find bugs e.g. annotation errors.
   d. Do you need more data?
2. **Advice #7:** Be cautious and patient
   a. You generally don't know in advance how good the results will be.
   b. If you get a good result, check it twice before announcing it.
   c. If you get a bad result, check it twice before discarding it.
   d. Be patient, but don't lose sight of your objective (it is easy to get lost)
3. **Advice #8:** ML is conceptually simple, don't let yourself drown in applied complexity
   a. Keep, for you and others, a clear overview of what is being done and what are the possible next steps.
   b. Make it easy to iterate and tests new models.

Advices for applied ML development (4/4)

1. **Advice #9:** Write unit tests!
   a. It will save your time even in the short term.
   b. Help you think about the data.
   c. E2e tests are good, but do not replace unit tests.
2. **Advice #10:** Be curious
   a. Allocate time to discover new tools and test new ideas.
   b. Does not have to be related to your project.
   c. [Exploration Friday]

Flexible & extendable ML with the JAX library

JAX

- Array-based numerical computation in Python
- Think: Numpy + utility to compute gradients + GPU / TPU support + distributed.
- Not specific to neural networks but work very well with them (easy to implement backprop).

```
import jax

def f(x):
 return x**2 + 5

f(2.0) # Note: 2² + 5 = 9
>> 9.

f(jax.numpy.array([6.0, 9.0]))
>> [41., 86.]

df = jax.grad(f)
df(2.0)
# Note: d(x² + 5)/dx = 2x = 4
>> 4.
```

Deep learning with JAX

- **Particularity #1 of JAX:** Pure functional API i.e. no internal state.

```
model = Model(...)
model.update(data)
model.predict(data)
```

```
params = {...}
params = update(params, data)
predict(params, data)
```

- **Con**:
  - More verbose than alternatives (e.g. TensorFlow, Keras, PyTorch)
- **Pro**:
  - No magic / hidden behavior => no surprises, easy to debug.
  - Easy to extend
  - Easy to implement novel ideas.
- All DL research at Google is done with JAX. Gemini & gemma use JAX.

Deep learning with JAX

- **Particularity #2 of JAX:**
  - Only contain array computation primitives.
  - No neural network specific functionalities (e.g., no backpropagation, losses, layers, scheduling, or dataset concepts).
- Instead, user are expected to **implement things manually** or use **JAX-side libraries**.
  - FLAX: Layers e.g. attention
  - Optax: Loss functions and optimizers
  - Orbax: Checkpoints
  - JRaph: Graph Neural net
  - YDF: Decision Forests
- **Con:**
  - No centralized documentation; Multiple ways to do each thing.
- **Pro:**
  - JAX is small and can be learned in a few hours.
  - JAX is developed and maintained by a small and efficient team (high quality, low risk of complexity collapse).
  - Easy to connect with other python libraries tools.
  - Healthy competition for other groups to propose tools on top of JAX.
  - Can be used for other types of optimization / learning

Gradient Descent with JAX

```python
import jax

# We want to find x that minimize "loss_fn".
def loss_fn(x):
 return x ** 2

# Initial guess for x
x = jax.numpy.array(2.0)
print("Initial x: ", x)

# Gradient of loss_fn according to x
gradient_fn = jax.grad(loss_fn)

# Gradient descent algorithm
learning_rate = 0.2
num_iterations = 10
for _ in range(num_iterations):
 loss = loss_fn(x)
 gradient = gradient_fn(x)
 x =  x - learning_rate * gradient
 print(f"x:{x:.5f} loss:{loss:.5f} gradient:{gradient:.5f}")

print("Final x: ", x)
```

```
Initial x:  2.0
x:1.20000 loss:4.00000 gradient:4.00000
x:0.72000 loss:1.44000 gradient:2.40000
x:0.43200 loss:0.51840 gradient:1.44000
x:0.25920 loss:0.18662 gradient:0.86400
x:0.15552 loss:0.06718 gradient:0.51840
x:0.09331 loss:0.02419 gradient:0.31104
x:0.05599 loss:0.00871 gradient:0.18662
x:0.03359 loss:0.00313 gradient:0.11197
x:0.02016 loss:0.00113 gradient:0.06718
x:0.01209 loss:0.00041 gradient:0.04031
Final x:  0.012093236
```

Example: Ingredients to train a neural net with JAX

A way to feed training data

- Numpy & python generator works well if the data fits in memory.
- **Alt:** TensorFlow Dataset has pre-canned dataset and works well on large datasets.

A loss

- Optax is the de facto solution.
- **Alt:** Simple and research losses can be written by hand.

A way to backup model during training a.k.a. checkpoint [Skipped]

- Orbax is the de facto solution.
- Needed if training is long, likely to crash, or has expensive multi-steps.

A way to save the model [Skipped]

- Use checkpoints
- **Alt:** TensorFlow SavedModel

Scheduler [Skipped]

- Flax scheduling is the de facto solution.

Monitoring & early stopping [Skipped]

- Manually works well.

Model organization

- Manually works well for small models.
- **Alt:** Flax is the de facto solution for real development.

Learning a MLP with JAX [simplified + manual way]

```python
[some imports]

def mlp(params, x): # The model prediction
 x = jax.nn.relu(jnp.dot(x, params['w1']))
 x = jnp.dot(x, params['w2'])
 return x

def get_batches(batches=10, batch_size=100,
dims=10): # Gen random data
 for i in range(batches):
   x = np.random.randn(batch_size, dims)
   y = (np.sum(x, axis=1) > 0).astype(int)
   yield x, y

def loss_fn(params, x, y): # Objective
 logits = mlp(params, x)
 return optax.sigmoid_binary_cross_entropy(
          logits, y).mean()

# Function value and its derivative together.
loss_and_grads_fn =
 jax.value_and_grad(loss_fn)
```

```python
# Initial model parameter values
params = {
     'w1': jax.random.normal(
          jax.random.PRNGKey(1), (10, 10)),
     'w2': jax.random.normal(
          jax.random.PRNGKey(2), (10))}

@jax.jit
def train_step(params, x, y, lr=0.1):
 loss, grads = loss_and_grads_fn(
     params, x, y)
 new_params = jax.tree.map(
     lambda p, g: p - lr * g, params, grads)
 return new_params, loss


# Training loop
for epoch in range(10):
 for x, y in get_batches():
   params, loss = train_step(params, x, y)
 print(f'Epoch: {epoch} Loss: {loss:.4f}')
```

Einstein summation

- A compact & powerful way to write products + reduction operations on high dimensional arrays with ignored or re-ordered dimensions.
- Espetially convenient with transformers design.
- Not specific to JAX (available in Numpy, PyTorch, TF, Keras)
- **Syntax**

```
jax.numpy.einsum(<string instructions>, <operants...>)
```

- **Simple example: Matrix multiplication**

$$r[i,j] = \sum_k x[i,k]y[k,j]$$

```
r = jnp.einsum('ik,kj->ij',
    [[1, 2],[3, 4]],
    [[5, 6],[7, 8]])
```

input

, = *+

output

operant 1

operant 2

Einstein summation

Classical operations (no need for einsum in practice)

Vector product	z = \sum_i x[i] y[i]	`z = jnp.einsum('i,i->', x, y)`
Outer product	z[i,j] = x[i] y[j]	`z = jnp.einsum('i,j->ij', x, y)`
Transpose	z[i,j] = x[j,i]	`z = jnp.einsum('ij->ji', x)`
Matrix mult	z[i,j] = \sum_k x[i,k] y[k,j]	`z = jnp.einsum('ik,kj->ij', x, y)`
Sum	z = \sum_i x[i]	`z = jnp.einsum('i->', x)`

Batches operations

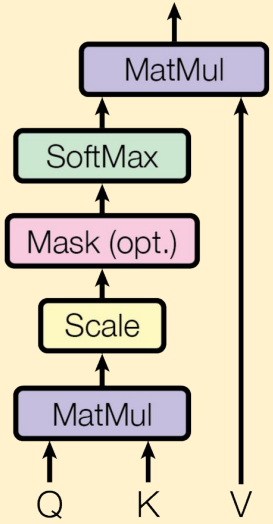
Vector product per batch	z[b] = \sum_i x[b,i] y[b,i]	`z = jnp.einsum('bi,bi->b', x, y)`
Sum, per batch	z[b] = \sum_i x[b,i]	`z = jnp.einsum('bi->b', x, y)`

Einstein summation

- Letters don't matter. Common notation: b: batch, d:dimension, s:sequence

Scaled Dot-Product Attention

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

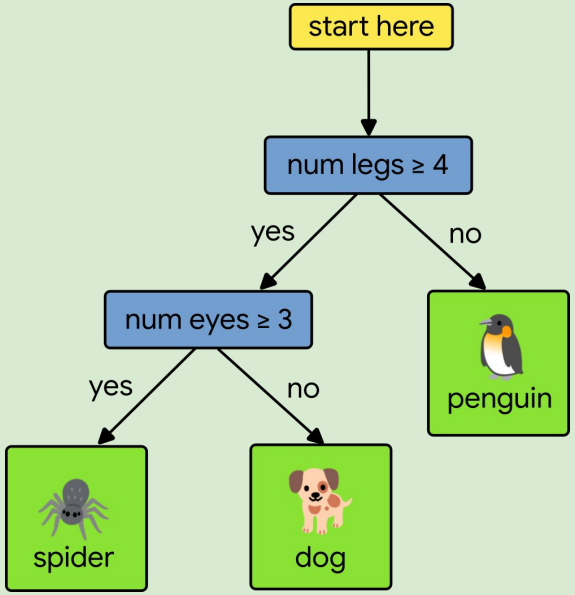



```
from jax.random import uniform, key
q = uniform(key(0), shape=[2,3,4])  # [b,s,d]
k = uniform(key(1), shape=[2,3,4])  # [b,s,d]
v = uniform(key(2), shape=[2,3,4])  # [b,s,d]

n_dim = q.shape[-1]
s = jnp.einsum('bid,bjd->bij', q, k ) # [b,s,s]
a = jax.nn.softmax(s / jnp.sqrt(n_dim), axis=-1) # [b,s,s]
jnp.einsum('bid,bdj->bij', a, v) # [b,s,d]
```

🌲 Gradient Boosted Trees with the YDF library

What are decision trees?

- Informally, a **decision tree** (DT) is a **multidimensional auto-histogram**.
- Formally, a DT is a **recursive partitioning** of the feature space.
- Each leaf node contains a prediction (green)
- Often, conditions are binary (two outcomes) and on a single attribute (axis aligned).
- Often, trained with greedy + divide and conquer algorithms.



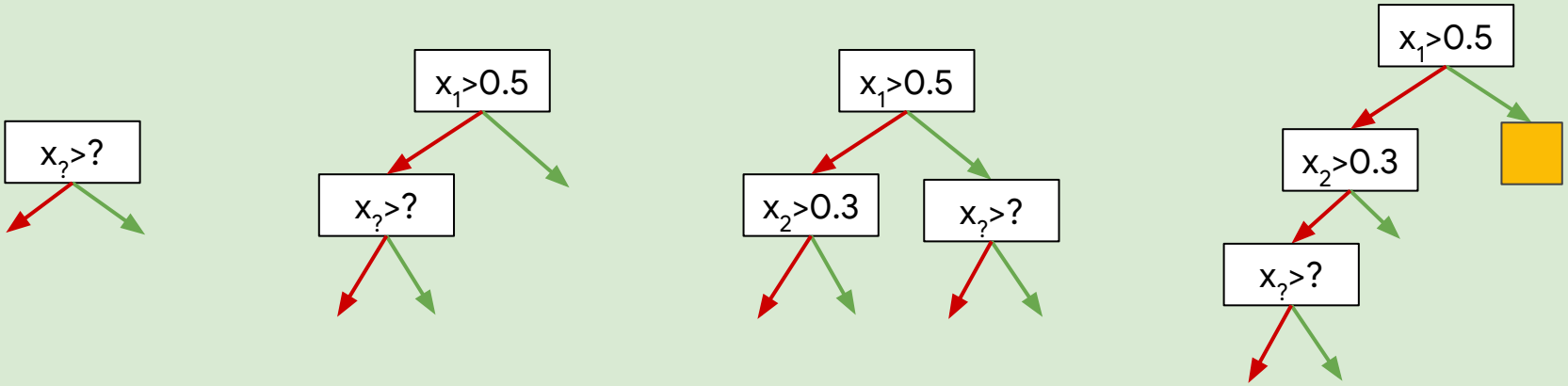
What are decision trees?

- **Benefits**
  - Don't need a lot of data.
  - No need for data normalization (e.g. z-score)
  - Natively consume numerical and categorical data.
  - Fast to train, very fast to run
  - Lot of useful capabilities (XAI, example distance, feature interaction).

- **Drawbacks**
  - Hard to balance underfitting / overfitting 🙁

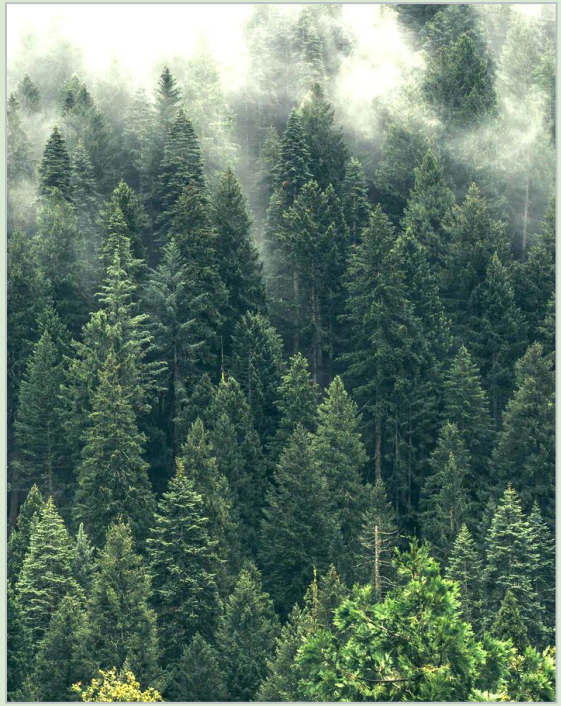
How do decision tree train?

- Often, trees are trained with a **greedy divide** and **conquer algorithm**.
- Main challenge: How can you find the best split efficiently?
- Famous learning algorithms: CART, ID3.



What are decision **forests**?

- A **decision forest** (DF) is a collection of DTs.
- Inherits all the good properties of decision trees.
- Solves the **DT quality problem** (SOTA on tabular data)
- Most common learning algorithms
  - **Random Forest:** Ensemble of decision trees trained with noise.
  - **Gradient Boosted Trees**: train each tree to predict & correct the error of the previous ones.
  - **AdaBoost:** Train trees in sequence. Increase the weights of poorly predicted examples.
- Exotic algorithms: Dart, Extremely randomized trees, sparse oblique forests



YDF

- Google's library to train decision forests.
- Objective:
  - Train, evaluate, understand and productionize a great model in 5 lines.
  - Everything is automated (but can be overridden).
  - Hard for the user to make mistakes.
  - Fast inference
  - Connects well with other ML tools (à-la-JAX).
- Implement both original and latest decision forests algorithms.
- Productionized in 2018 in Google
  - Train XXXk models per day
  - Used XXXM of times every second
- Powers the TensorFlow Decision Forests library

YDF

Yggdrasil Decision Forests

YDF: Basic usage

```
import pandas as pd
train_ds = pd.read_csv("train.csv")
test_ds = pd.read_csv("test.csv")
```

A dataset with numerical, categorical and missing values

age	workclass	fnlwgt	education	education_num	marital_status
44	Private	228057	7th-8th	4	Married-civ-spouse
20	Private	299047	Some-college	10	Never-married
40	Private	342164	HS-grad	9	Separated
30	Private	361742	Some-college	10	Married-civ-spouse
67	Self-emp-inc	171564	HS-grad	9	Married-civ-spouse

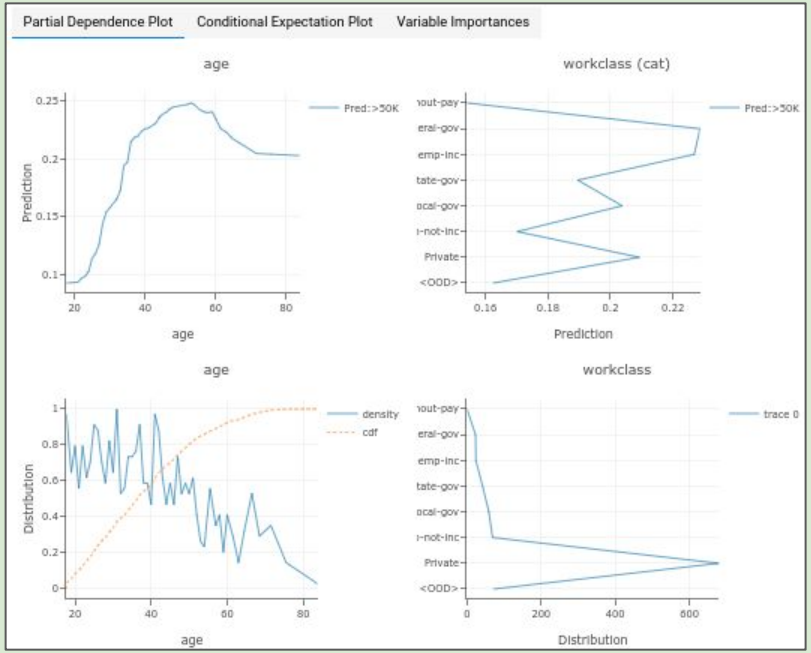
```
!pip install ydf
import ydf

# Train a model
learner =
    ydf.GradientBoostedTreesLearner(label="label")
model = learner.train(train_ds)

# Look at the model (e.g. input features,
# training logs, structure)
model.describe()

# Evaluate model
model.evaluate(test_ds)

# Make predictions
model.predict(test_ds)
```

Automatic model configuration and feature ingestion

Html report (in colab) + programmatic access

YDF: Understand models

```
model.analyze(test_ds)
model.analyze_prediction(test_ds.iloc[:1])
model.distance(test_ds, test_ds)
model.benchmark(test_ds)
```




```
Inference time per example and per cpu core: 0.891 us (microseconds)
Estimated over 345 runs over 3.004 seconds.
```

YDF: Export model to other tools

```
model.to_tensorflow_saved_model("tf_model", mode="tf")
model.to_tensorflow_function()
model.to_jax_function()
model.to_docker("tf_model")
model.to_cpp()
...
```

Bonus: Running model on FPGA with Conifer

```
import conifer

hls_cfg = conifer.backends.xilinxhls.auto_config()
hls_model = conifer.converters.convert_from_ydf(model, hls_cfg)
hls_model.compile()
```

Problems solved by decision forests

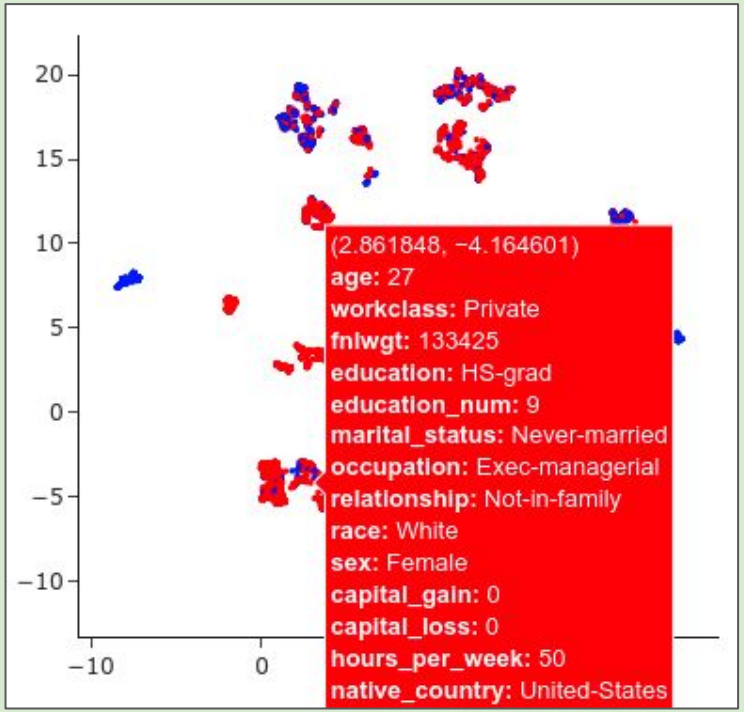
- **Classification**
  - **Output:** Probability of each possible output class
  - **Example:** Is this email a spam? What is the topic of this paper?
  - **ydf.Task.CLASSIFICATION**
- **Regression**
  - **Output:** Most likely numerical value.
  - **Example:** How long this experiment last?
  - **ydf.Task.REGRESSION**
- **Ranking**
  - **Output:** Sort values.
  - **Example:** Sort those webpages in order of interest.
  - **ydf.Task.RANKING**
- **Anomaly Detection**
  - **Output:** How normal / anormal is this value?
  - **Example:** Detect frauds or attacks. Find novelty.
  - **ydf.Task.ANOMALY_DETECTION**
- **Uplifting**
  - **Output:** How much a quantity would change if I did a specific action?
  - **Example:** How much the health of a person will improve if they receive a specific medical treatment? How much more likely this person will buy my product if they see my Ad?
  - **ydf.Task.CATEGORICAL_UPLIFT** or **ydf.Task.NUMERICAL_UPLIFT**

Bonus: Distance learning

- A decision forest can compute a distance between two observations.
- Even if observations have a mix of numerical, and categorical values, or values with different dimensionalities).

```
... = model.distance(data)
... = model.distance(data1, data2)
```

- Usage
  - How similar are two observations?
    - Look at the distance.
  - What are other similar observations?
    - Find the closest observations.
  - What are the clusters of observations?
    - Apply a clustering algorithm on the distances (e.g., sklearn.cluster. AgglomerativeClustering)
  - How are observations structured?
    - Apply a manifold learning algorithm on the distances (e.g. TSNE, UMAP)



Some decision forest training libraries

- **R Random Forest (2002)**: Easy to use implementation of the original Random Forest algorithm.
- **Scikit-learn (2007)**: Simplified original implementation of most major DF algorithms (Random Forest, Gradient Boosted Trees, AdaBoost).
- **XGBoost (2014)**: Popular modification of the Friedman's Gradient Boosted Trees algorithm.
- **LightGBM (2016; Microsoft)**: Same as XGBoost, but with more features.
- **CatBoost (2017; Yandex)**: Same as XGBoost, but with more features.
- **YDF / TensorFlow Decision Forests (2018, open source in 2021; Google)**: Original and state-of-the-art implementations for Random Forest and Gradient Boosted Trees.

Practical

Practical

- Two tutorial notebooks:
  - Practical ML with Google's tools | YDF
  - Practical ML with Google's tools | JAX
- The YDF one is easy. The JAX one is easy go get lost in => Start with YDF
- We are here if you have questions / issues / remarks