Probabilistic Programming with Deep Neural Networks

Andrés Masegosa

Department of Mathematics
University of Almería
Spain

https://andresmasegosa.github.io/
https://pgmlab.github.io/

This material jointly made with Thomas D. Nielsen (AAU).

Plan for this week

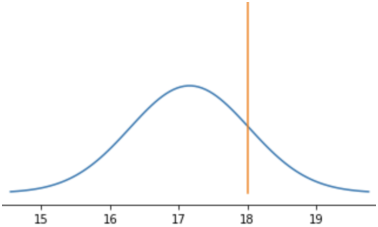
Day 1: Introduction to probabilistic programming languages

- Why do we need PPLs?
- Probabilistic programming in Pyro
- Hand-on exercises:
  - Probability Distributions in Pyro.
  - Probabilistic Models in Pyro.

Day 2: Probabilistic Models with Deep Neural Networks

- Uncertainty in Machine Learning
- Variational Inference
- Supervised/Unsupervised Learning
- Hand-on exercises
  - Bayesian Neural Networks
  - Variational Auto-encoders

What is a PPL?



Normal Distribution

What is a probabilistic model?

VQ-VAE-2

What is a PPL?

```python
import tensorflow as tf
import tensorflow_probability as tfp

# Pretend to load synthetic data set.
features = tfp.distributions.Normal(loc=0., scale=1.).sample(int(100e3))
labels = tfp.distributions.Bernoulli(logits=1.618 * features).sample()

# Specify model.
model = tfp.glm.Bernoulli()

# Fit model given data.
coeffs, linear_response, is_converged, num_iter = tfp.glm.fit(
    model_matrix=features[:, tf.newaxis],
    response=tf.cast(labels, dtype=tf.float32),
    model=model)
# ==> coeffs is approximately [1.618] (We're golden!)
```

Probabilistic Programming Language (PPL)

- An attempt to **unify probabilistic modeling and general programming languages**.

What is a PPL?

```python
import tensorflow as tf
import tensorflow_probability as tfp

# Pretend to load synthetic data set.
features = tfp.distributions.Normal(loc=0., scale=1.).sample(int(100e3))
labels = tfp.distributions.Bernoulli(logits=1.618 * features).sample()

# Specify model.
model = tfp.glm.Bernoulli()

# Fit model given data.
coeffs, linear_response, is_converged, num_iter = tfp.glm.fit(
    model_matrix=features[:, tf.newaxis],
    response=tf.cast(labels, dtype=tf.float32),
    model=model)
# ==> coeffs is approximately [1.618] (We're golden!)
```

Copy

Probabilistic Programming Language (PPL)

- An attempt to **unify probabilistic modeling and general programming languages**.
- A programming paradigm to define **general probabilistic models (mixing deterministic + stochastic functions)**.

What is a PPL?

```python
import tensorflow as tf
import tensorflow_probability as tfp

# Pretend to load synthetic data set.
features = tfp.distributions.Normal(loc=0., scale=1.).sample(int(100e3))
labels = tfp.distributions.Bernoulli(logits=1.618 * features).sample()

# Specify model.
model = tfp.glm.Bernoulli()

# Fit model given data.
coeffs, linear_response, is_converged, num_iter = tfp.glm.fit(
    model_matrix=features[:, tf.newaxis],
    response=tf.cast(labels, dtype=tf.float32),
    model=model)
# ==> coeffs is approximately [1.618] (We're golden!)
```

Probabilistic Programming Language (PPL)

- An attempt to **unify probabilistic modeling and general programming languages**.
- A programming paradigm to define **general probabilistic models (mixing deterministic + stochastic functions)**.
- Make probabilistic modeling more **applicable and powerful**.

Why PPLs?

Why PPLs?

Why do we need PPLs?

Why PPLs?

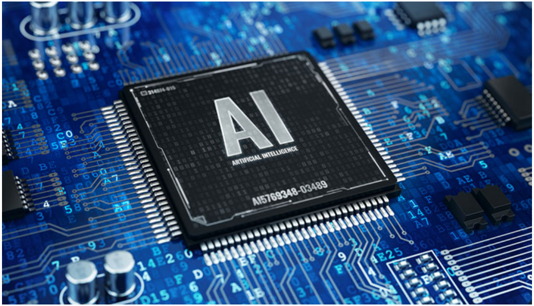
Why do we need PPLs?

- Reason 1: Try to democratize the development of AI systems.

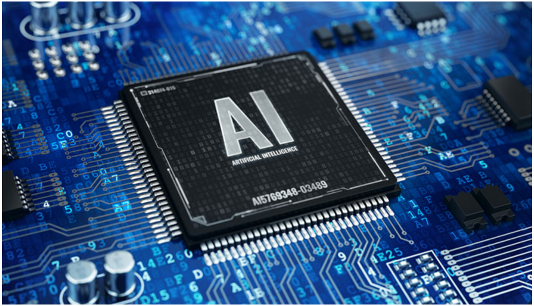
Why do we need PPLs?

- Reason 1: Try to democratize the development of AI systems.
- Reason 2: Try to make AI systems safer.

Reason 1: Try to democratize the development of AI systems.



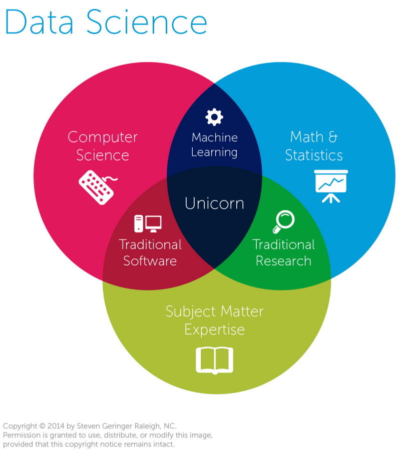
DARPA's Fund Call for PPLs in Artificial Intelligence.



DARPA's Fund Call for PPLs in Artificial Intelligence.

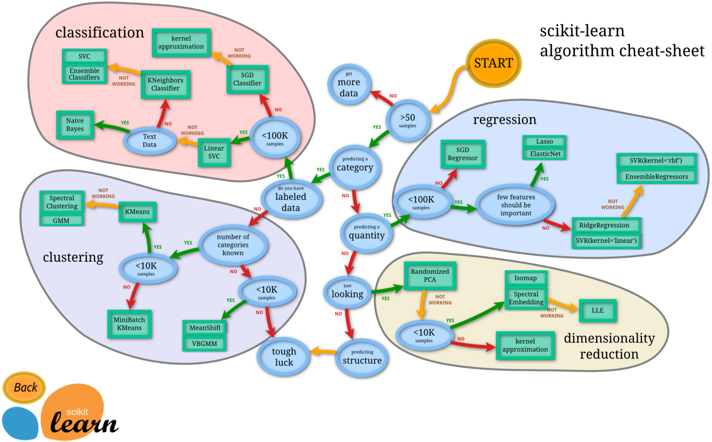
The development of machine learning systems requires enormous efforts.

Why PPLs?



The development of **machine learning systems** requires enormous efforts.

- **It requires of highly qualified experts.**



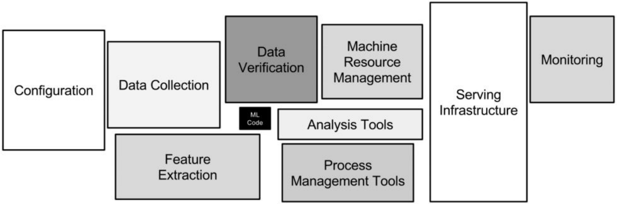
The development of machine learning applications requires enormous effort.

- It is necessary to have highly qualified experts.
- **It is difficult to find the ML model most suitable for an application.**

Why PPLs?

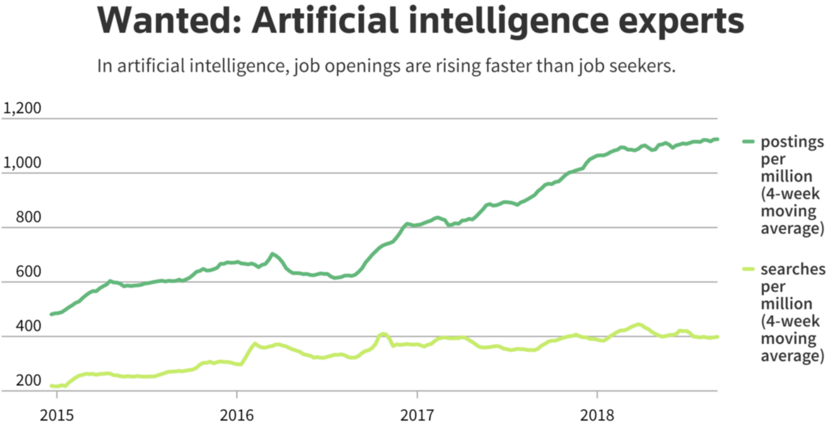
Hidden Technical Debt in Machine Learning Systems

D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips
{dsculley,gholt,dgg,edavydov,toddphillips}@google.com
Google, Inc.



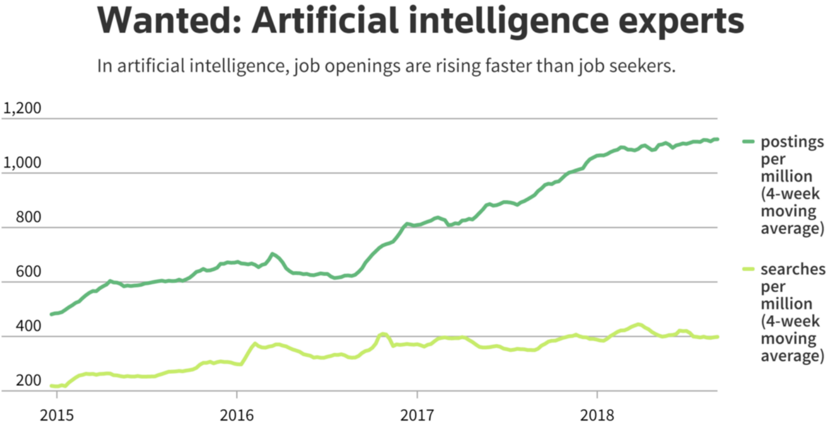
The development of machine learning applications requires enormous effort.

- It is necessary to have highly qualified experts.
- It is difficult to find the ML model most suitable for an application.
- **Programming a ML model is a complex task where many problems are intermingled**.



Consequences:

- Shortage of AI experts (and high salaries).



Consequences:

- Shortage of AI experts (and high salaries).
- Only big corporations have the resources for developing ML systems.

Why PPLs?

Similar situation than 50 years ago:

Why PPLs?

Similar situation than 50 years ago:

- People used to program in low-level programming languages.

Why PPLs?

Similar situation than 50 years ago:

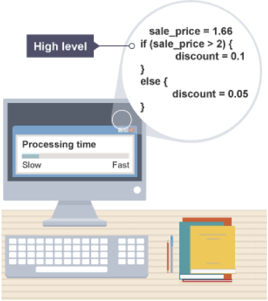
- People used to program in low-level programming languages.
- Programming was complex and demand high-expertise.

Why PPLs?

Similar situation than 50 years ago:

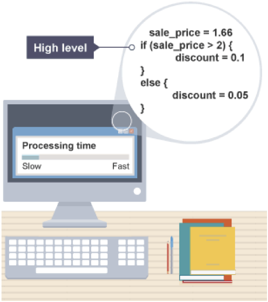
- People used to program in low-level programming languages.
- Programming was complex and demand high-expertise.
- Focus on application and low-level hardware details.

Why PPLs?



High-level programming languages brought many advantages:

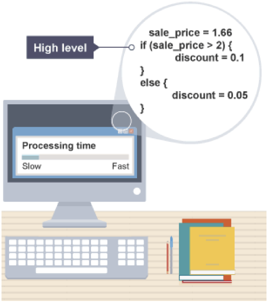
Why PPLs?



High-level programming languages brought many advantages:

- Programmers focused on the applications.

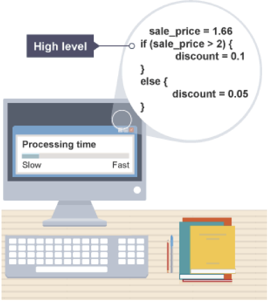
Why PPLs?



High-level programming languages brought many advantages:

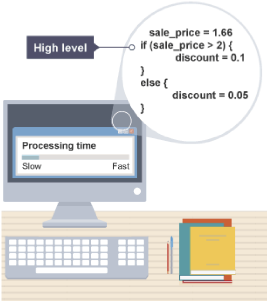
- Programmers focused on the applications.
- Hardware Experts focused on compilers.

Why PPLs?



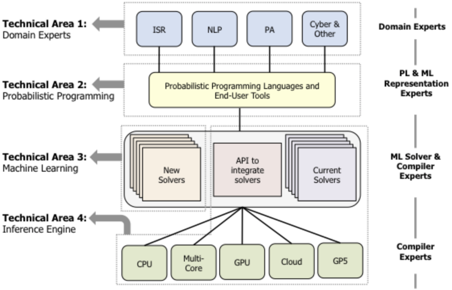
High-level programming languages brought many advantages:

- Programmers focused on the applications.
- Hardware Experts focused on compilers.
- High gains in productivity.



High-level programming languages brought many advantages:

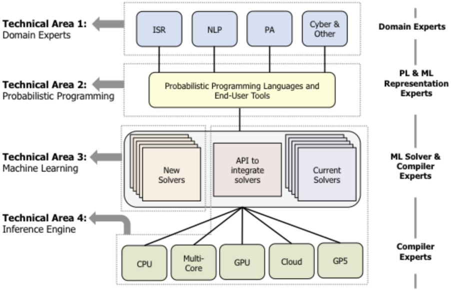
- Programmers focused on the applications.
- Hardware Experts focused on compilers.
- High gains in productivity.
- "Democratization" of the software development.



PPLs as high-level programming languages for **machine learning systems**:

- Stacked architecture

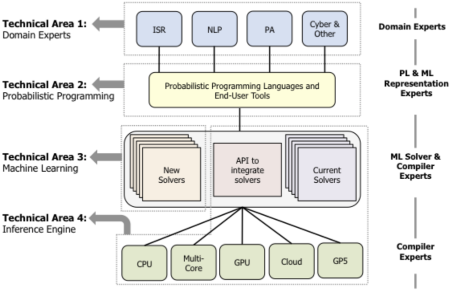
Why PPLs?



PPLs as high-level programming languages for **machine learning systems**:

- Stacked architecture
- Different Domain Experts will code their models using the same language.

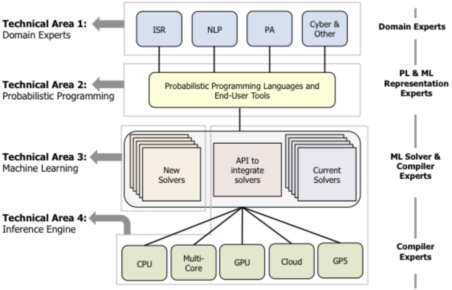
Why PPLs?



PPLs as high-level programming languages for **machine learning systems**:

- Stacked architecture
- Different Domain Experts will code their models using the same language.
- ML experts will focus on the development of new ML solvers.

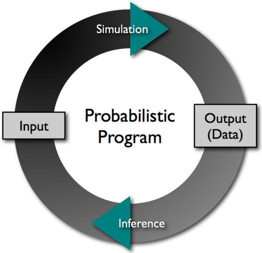
Why PPLs?



PPLs as high-level programming languages for **machine learning systems**:

- Stacked architecture
- Different Domain Experts will code their models using the same language.
- ML experts will focus on the development of new ML solvers.
- Compile experts will focus on running these ML solvers on specialized hardware.

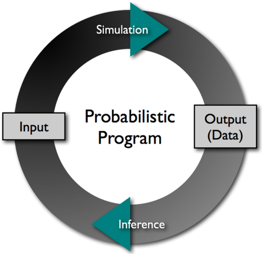
Why PPLs?



Benefits of PPLs:

- Simplify machine learning model code.

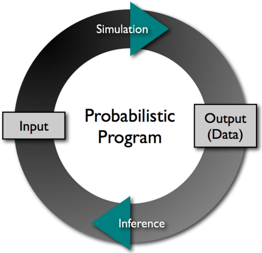
Why PPLs?



Benefits of PPLs:

- Simplify machine learning model code.
- Reduce development time and cost to encourage experimentation.

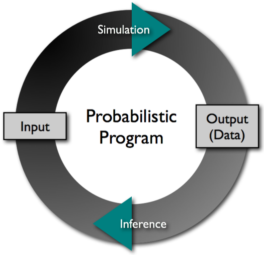
Why PPLs?



Benefits of PPLs:

- Simplify machine learning model code.
- Reduce development time and cost to encourage experimentation.
- Facilitate the construction of more sophisticated models.

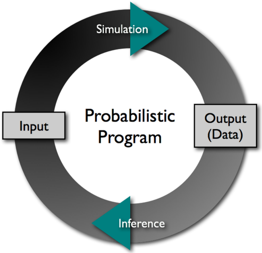
Why PPLs?



Benefits of PPLs:

- Simplify machine learning model code.
- Reduce development time and cost to encourage experimentation.
- Facilitate the construction of more sophisticated models.
- Reduce the necessary level of expertise.

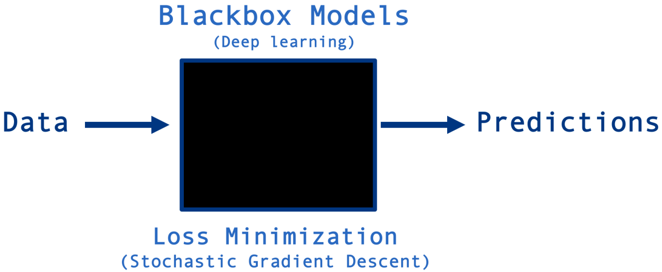
Why PPLs?



Benefits of PPLs:

- Simplify machine learning model code.
- Reduce development time and cost to encourage experimentation.
- Facilitate the construction of more sophisticated models.
- Reduce the necessary level of expertise.
- "Democratization" of the development of ML systems.

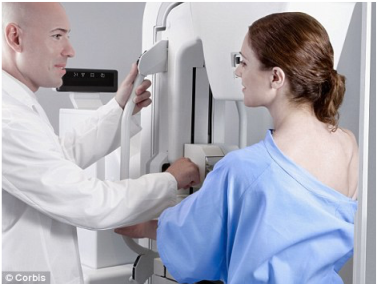
Reason 2: Try to make AI systems safer



Deep Learning Based AI Systems :

- **Issue 1**: Hard to **interpret**.
- **Issue 2**: No possible to know **how sure they are** in a particular prediction.
- Enormously **limit** the application of AI to many **real life problems**.

Today AI



AI in Health Care:

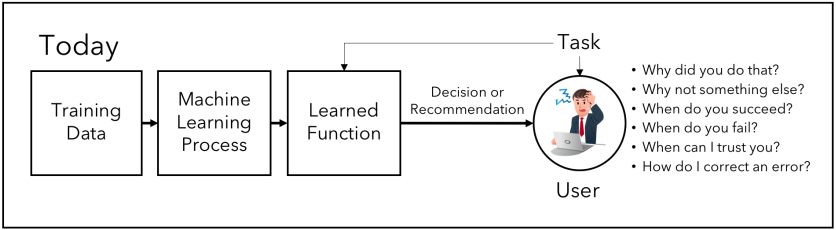
- Patients **need to know** why they are prescribed some treatment.
- An doctor can supervise the machine (**humans in the loop**).
- Extensible to any safe-critical system.

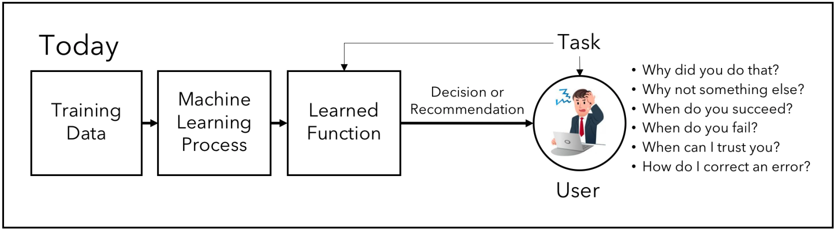
Today AI

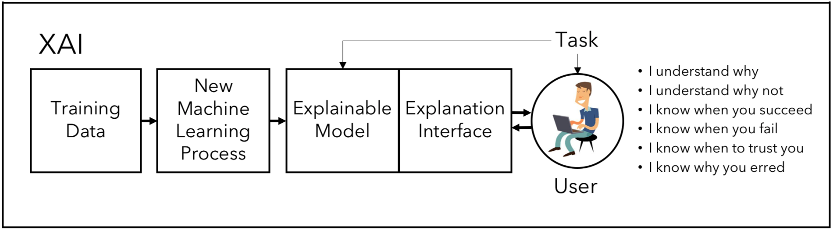
AI in Automated Systems:

- The system should detect when it is in a **completely new situation**.
- Let a **human take the control**.

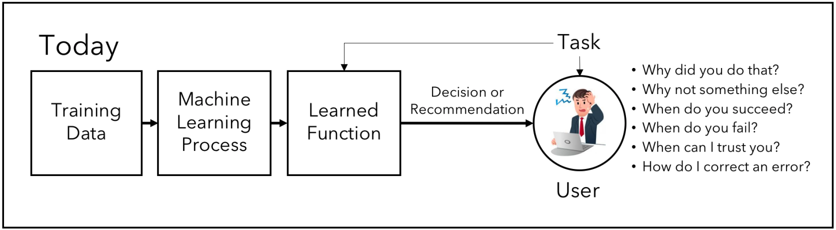
XAI: Explainable AI

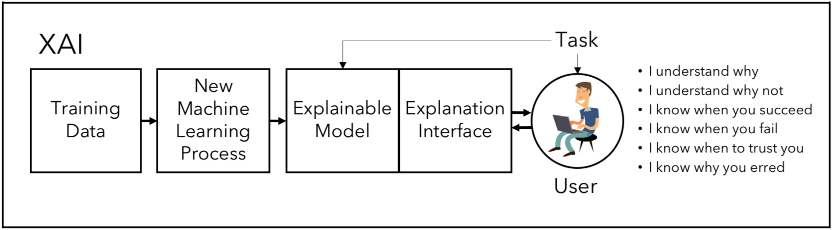



Today
Training Data
Machine Learning Process
Learned Function
Task
Decision or Recommendation
User
• Why did you do that?
• Why not something else?
• When do you succeed?
• When do you fail?
• When can I trust you?
• How do I correct an error?

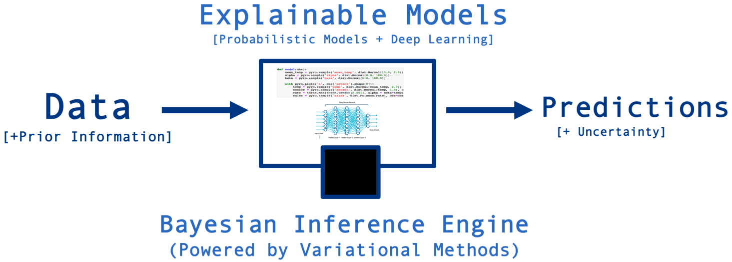

XAI
Training Data
New Machine Learning Process
Explainable Model
Explanation Interface
Task
User
• I understand why
• I understand why not
• I know when you succeed
• I know when you fail
• I know when to trust you
• I know why you erred

XAI: Explainable AI





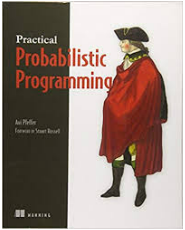
DARPA's Fund Call for XAI projects.



PPL based Systems :

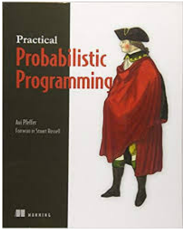
- Addressing Issue 1: PPLs provide **transparent** model description.
- Addressing Issue 2: PPLs provide **uncertainty estimation** both in models and predictions.

Brief Historical Review of PPLs



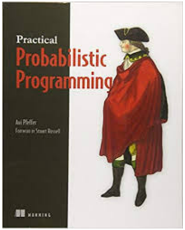
1st Generation of PPLs :

- Bugs, WinBugs, Jags, Figaro, etc.



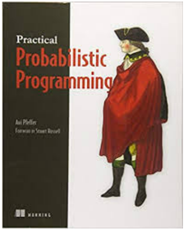
1st Generation of PPLs :

- Bugs, WinBugs, Jags, Figaro, etc.
- **Turing-complete** probabilistic programming languages. (i.e. they can represent any computable probability distribution).



1st Generation of PPLs :

- Bugs, WinBugs, Jags, Figaro, etc.
- **Turing-complete** probabilistic programming languages. (i.e. they can represent any computable probability distribution).
- Inference engine based on **Monte Carlo** methods.



1st Generation of PPLs :

- Bugs, WinBugs, Jags, Figaro, etc.
- **Turing-complete** probabilistic programming languages. (i.e. they can represent any computable probability distribution).
- Inference engine based on **Monte Carlo** methods.
- They did **not scale** to large data samples/high-dimensional models.

PPLs

2nd Generation of PPLs :

- Infer.net, Factorie, Amidst, etc.

2nd Generation of PPLs :

- Infer.net, Factorie, Amidst, etc.
- Inference engine based on **message passage algorithms** and/or variational inference methods.

2nd Generation of PPLs :

- Infer.net, Factorie, Amidst, etc.
- Inference engine based on **message passage algorithms** and/or variational inference methods.
- They did **scale** to large data samples/high-dimensional models.

2nd Generation of PPLs :

- Infer.net, Factorie, Amidst, etc.
- Inference engine based on **message passage algorithms** and/or variational inference methods.
- They did **scale** to large data samples/high-dimensional models.
- **Restricted** probabilistic model family (i.e. factor graphs, conjuage exponential family, etc.)

3rd Generation of PPLs :

- TensorFlow Probability, Pyro, PyMC3, InferPy, etc.

PPLs

3rd Generation of PPLs :

- TensorFlow Probability, Pyro, PyMC3, InferPy, etc.
- **Black Box Variational Inference** and Hamiltonian Monte-Carlo.

3rd Generation of PPLs :

- TensorFlow Probability, Pyro, PyMC3, InferPy, etc.
- **Black Box Variational Inference** and Hamiltonian Monte-Carlo.
- They did **scale** to large data samples/high-dimensional models.

3rd Generation of PPLs :

- TensorFlow Probability, Pyro, PyMC3, InferPy, etc.
- **Black Box Variational Inference** and Hamiltonian Monte-Carlo.
- They did **scale** to large data samples/high-dimensional models.
- **Turing-complete** probabilistic programming languages.

3rd Generation of PPLs :

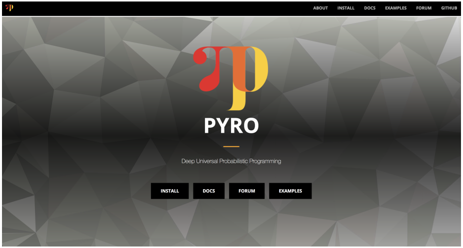
- TensorFlow Probability, Pyro, PyMC3, InferPy, etc.
- **Black Box Variational Inference** and Hamiltonian Monte-Carlo.
- They did **scale** to large data samples/high-dimensional models.
- **Turing-complete** probabilistic programming languages.
- Allow the inclusion of **deep neural networks**.

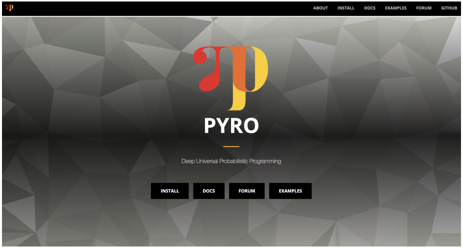
3rd Generation of PPLs :

- TensorFlow Probability, Pyro, PyMC3, InferPy, etc.
- **Black Box Variational Inference** and Hamiltonian Monte-Carlo.
- They did **scale** to large data samples/high-dimensional models.
- **Turing-complete** probabilistic programming languages.
- Allow the inclusion of **deep neural networks**.
- Rely on **deep learning frameworks** (TensorFlow, Pytorch, Theano, etc).

Pyro

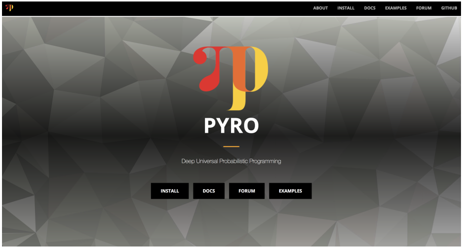
Pyro


ABOUT
INSTALL
DOCS
EXAMPLES
FORUM
GITHUB
PYRO
Deep Universal Probabilistic Programming
INSTALL
DOCS
FORUM
EXAMPLES



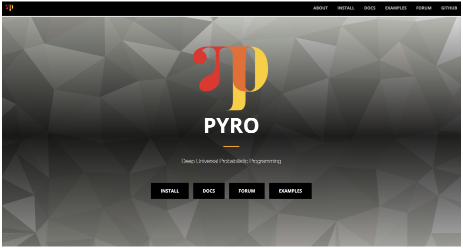
Pyro's main features (`www.pyro.ai`) :

- Developed by **UBER** (the car riding company).



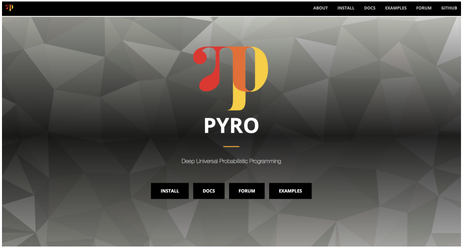
Pyro's main features (`www.pyro.ai`) :

- Developed by **UBER** (the car riding company).
- Focus on **probabilistic models with deep neural networks**.



Pyro's main features (`www.pyro.ai`) :

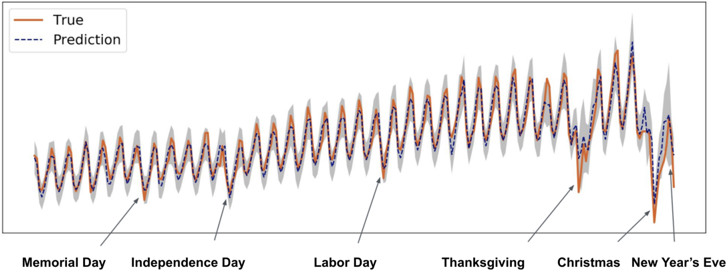
- Developed by **UBER** (the car riding company).
- Focus on **probabilistic models with deep neural networks**.
- Rely on **Pytorch** (Deep Learning Framework).



Pyro's main features (`www.pyro.ai`) :

- Developed by **UBER** (the car riding company).
- Focus on **probabilistic models with deep neural networks**.
- Rely on **Pytorch** (Deep Learning Framework).
- Enable **GPU** accelaration and distributed learning.

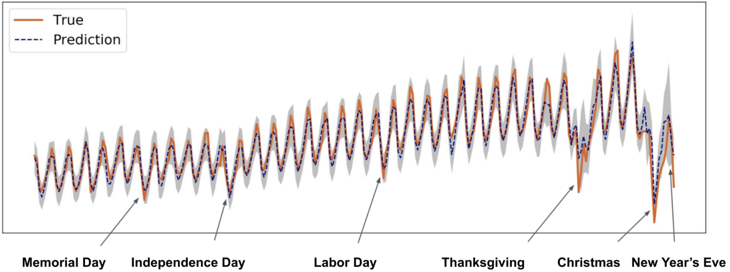
Pyro at UBER



Demand Prediction with Pyro:

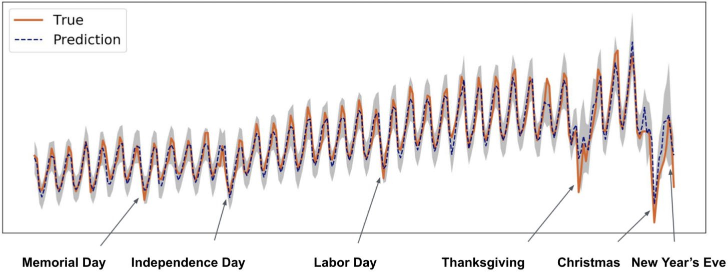
- Demand prediction is **critical** for user experience, resource allocation, etc.

Pyro at UBER



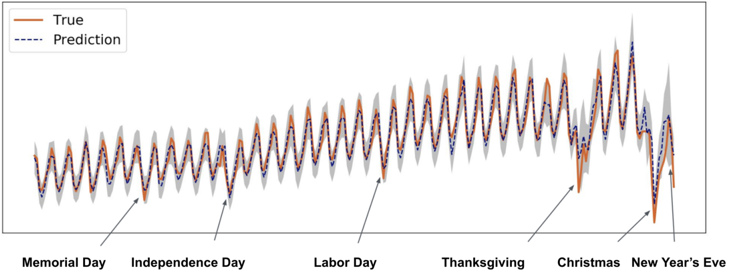
Demand Prediction with Pyro:

- Demand prediction is **critical** for user experience, resource allocation, etc.
- LSTM powerful for time series modelling.



Demand Prediction with Pyro:

- Demand prediction is **critical** for user experience, resource allocation, etc.
- LSTM powerful for time series modelling.
- Prediction at **special events** is challenging: weather, population growth, etc.



Demand Prediction with Pyro:

- Demand prediction is **critical** for user experience, resource allocation, etc.
- LSTM powerful for time series modelling.
- Prediction at **special events** is challenging: weather, population growth, etc.
- Bayesian LSTM provides **uncertainty estimation**.

Pyro's Distributions

```
In [3]: normal = dist.Normal(0,1)
        normal

Out[3]: Normal(loc: 0.0, scale: 1.0)
```

Pyro's distributions (http://docs.pyro.ai/en/stable/distributions.html) :

- Wide range of distributions: Normal, Beta, Cauchy, Dirichlet, Gumbel, Poisson, Pareto, etc.

```
In [15]: sample = normal.sample()
         sample

Out[15]: tensor(0.4908)
```

Pyro's distributions (http://docs.pyro.ai/en/stable/distributions.html) :

- Wide range of distributions: Normal, Beta, Cauchy, Dirichlet, Gumbel, Poisson, Pareto, etc.
- Samples from the distributions are Pytorch's Tensor objects (i.e. multidimensional arrays).

```
In [17]: sample = normal.sample(sample_shape=[3,4,5])
         sample.shape

Out[17]: torch.Size([3, 4, 5])
```

Pyro's distributions (`http://docs.pyro.ai/en/stable/distributions.html`) :

- Wide range of distributions: Normal, Beta, Cauchy, Dirichlet, Gumbel, Poisson, Pareto, etc.
- Samples from the distributions are Pytorch's Tensor objects (i.e. multidimensional arrays).

```
In [19]: torch.sum(normal.log_prob(sample))

Out[19]: tensor(-85.1003)
```

Pyro's distributions (`http://docs.pyro.ai/en/stable/distributions.html`) :

- Wide range of distributions: Normal, Beta, Cauchy, Dirichlet, Gumbel, Poisson, Pareto, etc.
- Samples from the distributions are Pytorch's Tensor objects (i.e. multidimensional arrays).
- Operations, like log-likelihood, are defined over tensors (with GPU acceleration powered by Pytorch).

```
In [9]: normal = dist.Normal(torch.tensor([1.,2.,3.]),1.)
        normal

Out[9]: Normal(loc: torch.Size([3]), scale: torch.Size([3]))
```

Pyro's distributions (http://docs.pyro.ai/en/stable/distributions.html) :

- Wide range of distributions: Normal, Beta, Cauchy, Dirichlet, Gumbel, Poisson, Pareto, etc.
- Samples from the distributions are Pytorch's Tensor objects (i.e. multidimensional arrays).
- Operations, like log-likelihood, are defined over tensors (with GPU acceleration powered by Pytorch).
- Multiple distributions can be embedded in single object (to define efficient vectorized operations).

```
In [10]: normal.sample()

Out[10]: tensor([2.0592, 2.4035, 3.1918])
```

Pyro's distributions (`http://docs.pyro.ai/en/stable/distributions.html`) :

- Wide range of distributions: Normal, Beta, Cauchy, Dirichlet, Gumbel, Poisson, Pareto, etc.
- Samples from the distributions are Pytorch's Tensor objects (i.e. multidimensional arrays).
- Operations, like log-likelihood, are defined over tensors (with GPU acceleration powered by Pytorch).
- Multiple distributions can be embedded in single object (to define efficient vectorized operations).

```
In [11]: normal.log_prob(normal.sample())
Out[11]: tensor([-0.9402, -1.2113, -2.3214])
```

Pyro's distributions (http://docs.pyro.ai/en/stable/distributions.html) :

- Wide range of distributions: Normal, Beta, Cauchy, Dirichlet, Gumbel, Poisson, Pareto, etc.
- Samples from the distributions are Pytorch's Tensor objects (i.e. multidimensional arrays).
- Operations, like log-likelihood, are defined over tensors (with GPU acceleration powered by Pytorch).
- Multiple distributions can be embedded in single object (to define efficient vectorized operations).

Now it is your turn!

Open the Notebook and Play around

- Test that you have installed the basic packages.
- Test that you can run the first lines of code.
- Play a bit with the code in Section 1 of the notebook.

`Day1/students_PPLs_Intro.ipynb`

`https://github.com/PGM-Lab/ASML-Tbilisi`

Pyro's Models

In [12]:
```python
def model():
    temp = pyro.sample('temp', dist.Normal(15.0, 2.0))
    return temp

print(model())
print(model())
```

```
tensor(12.3926)
tensor(22.5272)
```

Pyro's models (http://pyro.ai/examples/intro_part_i.html) :

- A probabilistic model is defined as a **stochastic function**.

```
In [12]: def model():
             temp = pyro.sample('temp', dist.Normal(15.0, 2.0))
             return temp

         print(model())
         print(model())

tensor(12.3926)
tensor(22.5272)
```

Pyro's models (http://pyro.ai/examples/intro_part_i.html) :

- A probabilistic model is defined as a **stochastic function**.
- Each random variable is associated to a **primitive stochastic function** using the construct **pyro.sample(...)**.

```
In [21]: def model():
             temp = pyro.sample('temp', dist.Normal(15.0, 2.0))
             sensor = pyro.sample('sensor', dist.Normal(temp, 1.0))
             return (temp, sensor)

         out1 = model()
         out1

Out[21]: (tensor(15.8576), tensor(16.9907))
```

Pyro's models (http://pyro.ai/examples/intro_part_i.html) :

- A **stochastic function** can be defined as a composition of **primitive stochastic functions**.

```
In [21]: def model():
             temp = pyro.sample('temp', dist.Normal(15.0, 2.0))
             sensor = pyro.sample('sensor', dist.Normal(temp, 1.0))
             return (temp, sensor)

         out1 = model()
         out1

Out[21]: (tensor(15.8576), tensor(16.9907))
```

Pyro's models (http://pyro.ai/examples/intro_part_i.html) :

- A **stochastic function** can be defined as a composition of **primitive stochastic functions**.
- We define the **joint probability distribution**:

$$p(\mathtt{sensor}, \mathtt{temp}) = p(\mathtt{sensor}|\mathtt{temp})p(\mathtt{temp})$$

Pyro's Inference

In [22]:
```python
#The observatons
obs = {'sensor': torch.tensor(18.0)}

def model(obs):
    temp = pyro.sample('temp', dist.Normal(15.0, 2.0))
    sensor = pyro.sample('sensor', dist.Normal(temp, 1.0),obs=obs['sensor'])
```

Pyro's inference (http://pyro.ai/examples/intro_part_ii.html) :

- We can introduce observations (e.g. sensor = 18.0).

```
In [27]: #Run inference
svi(model,guide,obs, plot=True)

#Print results
print("P(Temperature|Sensor=18.0) = ")
print(dist.Normal(pyro.param("mean").item(), pyro.param("scale").item()))
print("")
```

Pyro's inference (http://pyro.ai/examples/intro_part_ii.html) :

- We can introduce observations (e.g. sensor = 18.0).
- We can query the **posterior probability distribution**:

$$p(\texttt{temp}|\texttt{sensor} = 18) = \frac{p(\texttt{sensor} = 18|\texttt{temp})p(\texttt{temp})}{\int p(\texttt{sensor} = 18|\texttt{temp})p(\texttt{temp})d\texttt{temp}}$$

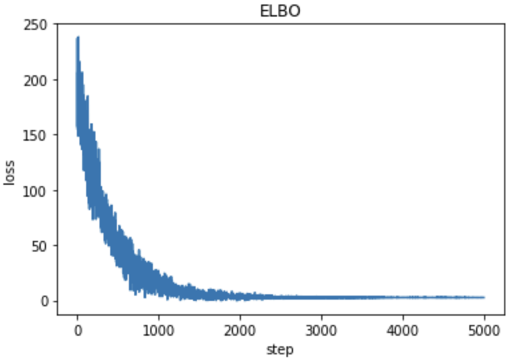
```
In [27]: #Run inference
svi(model,guide,obs, plot=True)

#Print results
print("P(Temperature|Sensor=18.0) = ")
print(dist.Normal(pyro.param("mean").item(), pyro.param("scale").item()))
print("")
```

Pyro's inference (http://pyro.ai/examples/intro_part_ii.html) :

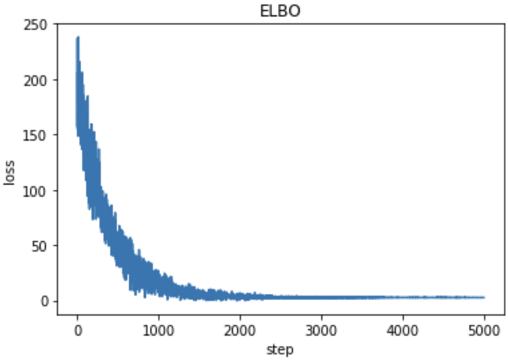
- We can introduce observations (e.g. sensor = 18.0).
- We can query the **posterior probability distribution**:

$$p(\texttt{temp}|\texttt{sensor} = 18) = \frac{p(\texttt{sensor} = 18|\texttt{temp})p(\texttt{temp})}{\int p(\texttt{sensor} = 18|\texttt{temp})p(\texttt{temp})d\texttt{temp}}$$

- Guide is an auxiliary method needed for inference (more details in the coming sessions).




```
P(Temperature|Sensor=18.0) =
Normal(loc: 17.39859390258789, scale: 0.9089401960372925)
```




```
P(Temperature|Sensor=18.0) =
Normal(loc: 17.39859390258789, scale: 0.9089401960372925)
```

Details on the inference method will be given on the following sessions.

Exercise 1: Change in the precision of the temperature sensor

- The precision of our temperature sensor is reflected in the **variance/scale** of the Normal distribution of the **sensor** variable.
- What happens if we get a **more precise temperature sensor**? Assume it has a variance/scale equal to 0.5.

`Day1/students_PPLs_Intro.ipynb`

`https://github.com/PGM-Lab/ASML-Tbilisi`

In [ ]:
```python
#The observatons
obs = {'sensor': torch.tensor([18., 18.7, 19.2, 17.8, 20.3, 22.4, 20.3, 21.2, 19.5, 20.1])}

def model(obs):
    for i in range(0,obs['sensor'].shape[0]):
        temp = pyro.sample(f'temp_{i}', dist.Normal(15.0, 2.0))
        sensor = pyro.sample(f'sensor_{i}', dist.Normal(temp, 1.0), obs=obs['sensor'][i])
```

Pyro's inference (http://pyro.ai/examples/intro_part_ii.html) :

- What if we have a **bunch of observations**, $\mathrm{s} = \{s_1, \ldots, s_n\}$

```
In [ ]: #The observatons
obs = {'sensor': torch.tensor([18., 18.7, 19.2, 17.8, 20.3, 22.4, 20.3, 21.2, 19.5, 20.1])}

def model(obs):
    for i in range(0,obs['sensor'].shape[0]):
        temp = pyro.sample(f'temp_{i}', dist.Normal(15.0, 2.0))
        sensor = pyro.sample(f'sensor_{i}', dist.Normal(temp, 1.0), obs=obs['sensor'][i])
```

Pyro's inference (`http://pyro.ai/examples/intro_part_ii.html`) :

- What if we have a **bunch of observations**, $\mathrm{s} = \{s_1, \ldots, s_n\}$
- A random variable is created for each observation (using a **for-loop**).

```
In [ ]: #The observatons
        obs = {'sensor': torch.tensor([18., 18.7, 19.2, 17.8, 20.3, 22.4, 20.3, 21.2, 19.5, 20.1])}

        def model(obs):
            for i in range(0,obs['sensor'].shape[0]):
                temp = pyro.sample(f'temp_{i}', dist.Normal(15.0, 2.0))
                sensor = pyro.sample(f'sensor_{i}', dist.Normal(temp, 1.0), obs=obs['sensor'][i])
```

Pyro's inference (http://pyro.ai/examples/intro_part_ii.html) :

- What if **we do not know the average temperature**?

```
In [28]: #The observatons
obs = {'sensor': torch.tensor([18., 18.7, 19.2, 17.8, 20.3, 22.4, 20.3, 21.2, 19.5, 20.1])}

def model(obs):
    mean_temp = pyro.param('mean_temp', torch.tensor(15.0))
    for i in range(0,obs['sensor'].shape[0]):
        temp = pyro.sample(f'temp_{i}', dist.Normal(mean_temp, 2.0))
        sensor = pyro.sample(f'sensor_{i}', dist.Normal(temp, 1.0), obs=obs['sensor'][i])
```

Pyro's inference (`http://pyro.ai/examples/intro_part_ii.html`) :

- What if **we do not know the average temperature**?
- We can introduce a **parameter** using **pyro.param** construct.

In [32]:
```
#Run inference
svi(model, guide, obs, num_steps=1000)

#Print results
print("Estimated Mean Temperature")
print(pyro.param("mean_temp").item())
```

```
Estimated Mean Temperature
19.129146575927734
```

Pyro's inference (`http://pyro.ai/examples/intro_part_ii.html`) :

- And **learn the parameter** with the same general inference algorithm.

$$\mu_t = \arg\max_{\mu} \ln p(s_1, \ldots, s_n | \mu)$$

- Details about the inference algorithm will be given in the next sessions.

```
In [28]: #The observatons
obs = {'sensor': torch.tensor([18., 18.7, 19.2, 17.8, 20.3, 22.4, 20.3, 21.2, 19.5, 20.1])}

def model(obs):
    mean_temp = pyro.param('mean_temp', torch.tensor(15.0))
    for i in range(0,obs['sensor'].shape[0]):
        temp = pyro.sample(f'temp_{i}', dist.Normal(mean_temp, 2.0))
        sensor = pyro.sample(f'sensor_{i}', dist.Normal(temp, 1.0), obs=obs['sensor'][i])
```

Pyro's inference (`http://pyro.ai/examples/intro_part_ii.html`) :

- What if **we want to capture uncertainty about the estimation of the average temperature**?

In [176]:
```python
#The observatons
obs = {'sensor': torch.tensor([18., 18.7, 19.2, 17.8, 20.3, 22.4, 20.3, 21.2, 19.5, 20.1])}

def model(obs):
    mean_temp = pyro.sample('mean_temp', dist.Normal(15.0, 2.0))
    for i in range(obs['sensor'].shape[0]):
        temp = pyro.sample(f'temp_{i}', dist.Normal(mean_temp, 2.0))
        sensor = pyro.sample(f'sensor_{i}', dist.Normal(temp, 1.0), obs=obs['sensor'][i])
```

Pyro's inference (`http://pyro.ai/examples/intro_part_ii.html`) :

- What if **we want to capture uncertainty about the estimation of the average temperature**?
- We can model this parameter with a **random variable**.

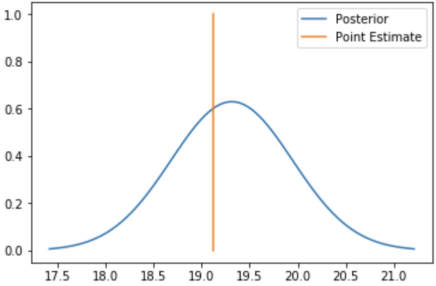
```
In [162]: import time

          #Run inference
          start = time.time()
          svi(model, guide, obs, num_steps=1000)

          #Print results
          print("P(mean_temp|Sensor=[18., 18.7, 19.2, 17.8, 20.3, 22.4, 20.3, 21.2, 19.5, 20.1]) =")
          print(dist.Normal(pyro.param("mean").item(), pyro.param("scale").item()))
          print("")
          end = time.time()
          print(f"{(end - start)} seconds")

          P(mean_temp|Sensor=[18., 18.7, 19.2, 17.8, 20.3, 22.4, 20.3, 21.2, 19.5, 20.1]) =
          Normal(loc: 19.199871063232422, scale: 0.6046891212463379)

          10.298431873321533 seconds
```

Pyro's inference (`http://pyro.ai/examples/intro_part_ii.html`) :

- And **learn the distribution** with the same general inference algorithm.

$$p(\mu_t | s_1, \ldots, s_{10})$$

- Details about the inference algorithm will be given in the next sessions.



f

Pyro's inference (`http://pyro.ai/examples/intro_part_ii.html`) :

- And **learn the distribution** with the same general inference algorithm.

$$p(\mu_t | s_1, \ldots, s_{10})$$

- Details about the inference algorithm will be given in the next sessions.

Defining Conditional Independences in Pyro

In [176]:
```
#The observatons
obs = {'sensor': torch.tensor([18., 18.7, 19.2, 17.8, 20.3, 22.4, 20.3, 21.2, 19.5, 20.1])}

def model(obs):
    mean_temp = pyro.sample('mean_temp', dist.Normal(15.0, 2.0))
    for i in range(obs['sensor'].shape[0]):
        temp = pyro.sample(f'temp_{i}', dist.Normal(mean_temp, 2.0))
        sensor = pyro.sample(f'sensor_{i}', dist.Normal(temp, 1.0), obs=obs['sensor'][i])
```

Pyro's cond. independeces (http://pyro.ai/examples/svi_part_ii.html) :

- Sensor variables are **independent** given temperature mean, μ_t.

$$p(s_1, t_1, ..., s_{10}, t_{10} | \mu_t) = \prod_{i=1}^{10} p(s_i, t_i | \mu_t)$$

In [37]:
```
#The observatons
obs = {'sensor': torch.tensor([18., 18.7, 19.2, 17.8, 20.3, 22.4, 20.3, 21.2, 19.5, 20.1])}

def model(obs):
    mean_temp = pyro.sample('mean_temp', dist.Normal(15.0, 2.0))
    with pyro.plate('a', obs['sensor'].shape[0]):
        temp = pyro.sample('temp', dist.Normal(mean_temp, 2.0))
        sensor = pyro.sample('sensor', dist.Normal(temp, 1.0), obs=obs['sensor'])
```

Pyro's cond. independeces (http://pyro.ai/examples/svi_part_ii.html) :

- Sensor variables are **independent** given temperature mean, μ_t.

$$p(s_1, t_1, ..., s_{10}, t_{10}|\mu_t) = \prod_{i=1}^{10} p(s_i, t_i|\mu_t)$$

- We can use **Pyro's plate construct** to introduce this independence.

```
In [165]: #Run inference
start = time.time()
svi(model, guide, obs, num_steps=1000)

#Print results
print("P(mean_temp|Sensor=[18., 18.7, 19.2, 17.8, 20.3, 22.4, 20.3, 21.2, 19.5, 20.1]) =")
print(dist.Normal(pyro.param("mean").item(), pyro.param("scale").item()))
print("")
end = time.time()
print(f"{(end - start)} seconds")

P(mean_temp|Sensor=[18., 18.7, 19.2, 17.8, 20.3, 22.4, 20.3, 21.2, 19.5, 20.1]) =
Normal(loc: 19.300748825073242, scale: 0.6379732489585876)

2.81210994720459 seconds
```

Pyro's cond. independeces (`http://pyro.ai/examples/svi_part_ii.html`) :

- We get large gains in efficiency due to **vectorized operations**.
- Execution time without *plate* is over 10s.

Now it is your turn!

- Exercise 2: The role of the **number of observations** in learning.
- Exercise 3: The role of the **prior distribution** in learning.

`Day1/students_PPLs_Intro.ipynb`

Toy Example: Ice-cream shop

Defining Machine Learning models with PPLs:

- We have an ice-cream shop and we record the ice-cream sales and the average temperature of the day.
- We know temperature affects the sales of ice-creams.
- We want to precisely find out how temperature affects ice-cream sales.

In [49]:
```
#The observatons
obs = {'sensor': torch.tensor([18., 18.7, 19.2, 17.8, 20.3, 22.4, 20.3, 21.2, 19.5, 20.1]),
       'sales': torch.tensor([46., 47., 49., 44., 50., 54., 51., 52., 49., 53.])}

def model(obs):
    mean_temp = pyro.sample('mean_temp', dist.Normal(15.0, 2.0))

    with pyro.plate('a', obs['sensor'].shape[0]):
        temp = pyro.sample('temp', dist.Normal(mean_temp, 2.0))
        sensor = pyro.sample('sensor', dist.Normal(temp, 1.0), obs=obs['sensor'])
```

Ice-cream Shop Model:

- We have observations from temperature and sales.

Pyro

In [43]:
```
#The observatons
obs = {'sensor': torch.tensor([18., 18.7, 19.2, 17.8, 20.3, 22.4, 20.3, 21.2, 19.5, 20.1]),
       'sales': torch.tensor([46., 47., 49., 44., 50., 54., 51., 52., 49., 53.])}

def model(obs):
    mean_temp = pyro.sample('mean_temp', dist.Normal(15.0, 2.0))

    with pyro.plate('a', obs['sensor'].shape[0]):
        temp = pyro.sample('temp', dist.Normal(mean_temp, 2.0))
        sensor = pyro.sample('sensor', dist.Normal(temp, 1.0), obs=obs['sensor'])
        sales = pyro.sample('sales', dist.Poisson(??????), obs=obs['sales'])
```

Ice-cream Shop Model:

- We have observations from temperature and sales.
- Sales are modeled with a **Poisson distribution**.

```
In [201]: #The observatons
obs = {'sensor': torch.tensor([18., 18.7, 19.2, 17.8, 20.3, 22.4, 20.3, 21.2, 19
       'sales': torch.tensor([46., 47., 49., 44., 50., 54., 51., 52., 49., 53.])

def model(obs):
    mean_temp = pyro.sample('mean_temp', dist.Normal(15.0, 2.0))
    alpha = pyro.sample('alpha', dist.Normal(0.0, 100.0))
    beta = pyro.sample('beta', dist.Normal(0.0, 100.0))

    with pyro.plate('a', obs['sensor'].shape[0]):
        temp = pyro.sample('temp', dist.Normal(mean_temp, 2.0))
        sensor = pyro.sample('sensor', dist.Normal(temp, 1.0), obs=obs['sensor']
        rate = torch.max(torch.tensor(0.001), alpha + beta*temp)
        sales = pyro.sample('sales', dist.Poisson(rate), obs=obs['sales'])
```

Ice-cream Shop Model:

- We have observations from temperature and sales.
- Sales are modeled with a **Poisson distribution**.
- The rate of the Poisson **linearly depends of the real temperature**.

In [48]:
```
#Run inference
svi(model, guide, obs, num_steps=1000)

#Print results
print("Posterior Temperature Mean")
print(dist.Normal(pyro.param("mean").item(), pyro.param("scale").item()))
print("")
print("Posterior Alpha")
print(dist.Normal(pyro.param("alpha_mean").item(), pyro.param("alpha_scale").item()))
print("")
print("Posterior Beta")
print(dist.Normal(pyro.param("beta_mean").item(), pyro.param("beta_scale").item()))
```

```
Posterior Temperature Mean
Normal(loc: 19.311052322387695, scale: 0.6258021593093872)

Posterior Alpha
Normal(loc: 19.773971557617188, scale: 1.8541947603225708)

Posterior Beta
Normal(loc: 1.5178951025009155, scale: 0.1155082955956459)
```

Ice-cream Shop Model:

- We run the **(variational) inference engine** and get the results.

In [48]:
```
#Run inference
svi(model, guide, obs, num_steps=1000)

#Print results
print("Posterior Temperature Mean")
print(dist.Normal(pyro.param("mean").item(), pyro.param("scale").item()))
print("")
print("Posterior Alpha")
print(dist.Normal(pyro.param("alpha_mean").item(), pyro.param("alpha_scale").item()))
print("")
print("Posterior Beta")
print(dist.Normal(pyro.param("beta_mean").item(), pyro.param("beta_scale").item()))
```

```
Posterior Temperature Mean
Normal(loc: 19.311052322387695, scale: 0.6258021593093872)

Posterior Alpha
Normal(loc: 19.773971557617188, scale: 1.8541947603225708)

Posterior Beta
Normal(loc: 1.5178951025009155, scale: 0.1155082955956459)
```

Ice-cream Shop Model:

- We run the **(variational) inference engine** and get the results.
- With PPLs, we only care about modeling, **not about the low-level details** of the machine-learning solver.

Exercise 4: Introduce Humidity in the Icecream shop model

- Assume we also have a bunch of humidity senor measurements.
- Assume the sales are also linearly influenced by the humidity.
- **Extend the above model in order to integrate all of that**.

`Day1/students_PPLs_Intro.ipynb`