INFERNO AS A DROP-IN* LOSS FUNCTION

Giles Strong

(INFERNO by Pablo de Castro & Tommaso Dorigo)

gradHEP Campfire, Online - 12/02/21

giles.strong@outlook.com

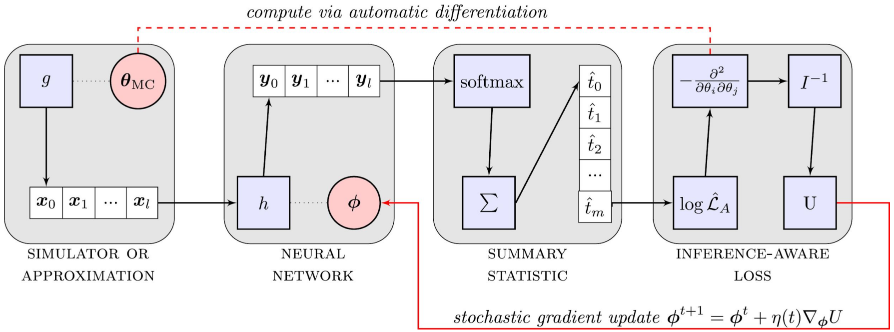
twitter.com/Giles_C_Strong

https://gilesstrong.github.io/website/

github.com/GilesStrong

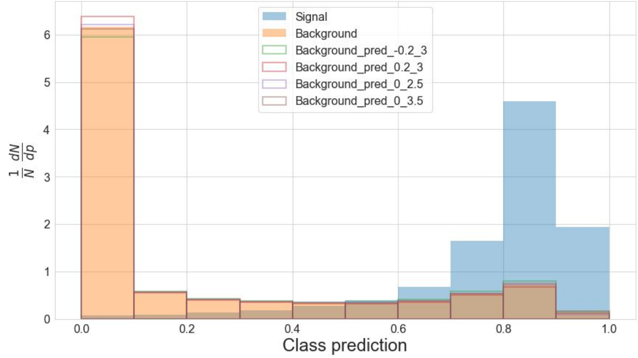
INFERNO

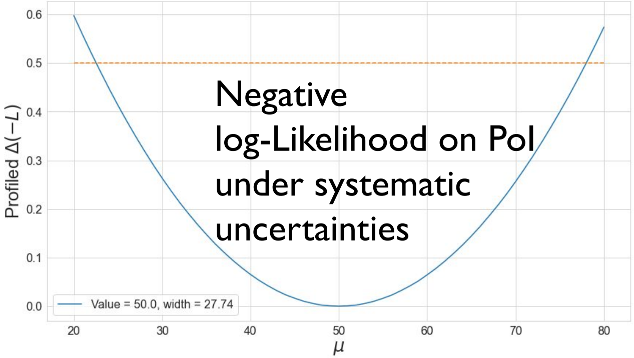
- de Castro & Dorigo 2018
- Directly optimise DNN for stat. Inf.
  - DNN output is binned summary statistic
    - Softmax output - can hard-assign after training
  - Loss is inversely proportional to the uncertainty on parameter of interest
    - Computed from inverted Hessian of likelihood w.r.t. parameters
    - Includes nuisances on both input features & normalisation
  - I.e. DNN encouraged to become sensitive to PoI and resilient to nuisances



INFERNO BENEFIT - TOY EXAMPLE

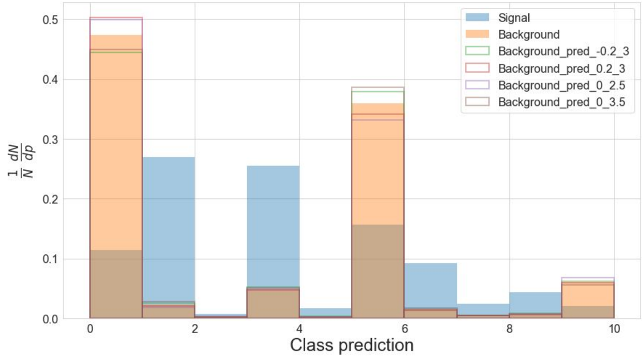
Binned output of binary cross-entropy classifier

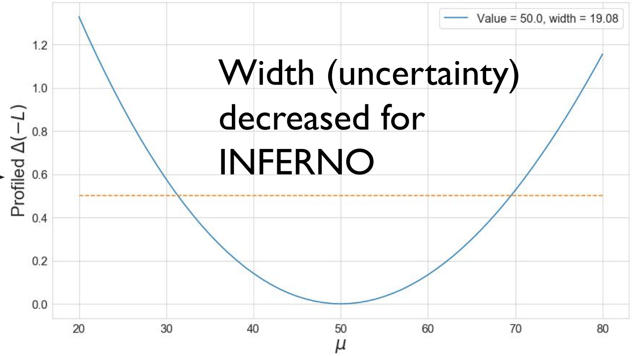




Negative log-Likelihood on PoI under systematic uncertainties

Hard-assigned output of INFERNO model





Width (uncertainty) decreased for INFERNO

PROCESS - ONE UPDATE STEP

1. Minibatch: x - inputs, y - targets
2. Cache tensor of nuisances at nominal values (zero)
3. Modify inputs according to shape nuisances: $x \leftarrow x + \text{nuisances}$
   a. E.g. $x_1 \leftarrow x_1 + 0$, $x_2 \leftarrow x_2 * (x_2 + 0)/x_2$,
   b. Nuisances don't change input values but allow gradients to be computed
   c. Instead add "the potential to be modified"
4. Pass x through NN (with softmax)
5. Predictions y_p: probabilities per bin/class
6. Index signal & background using y
   a. Sum counts per bin → sig & bkg shapes
   b. Detach/stop gradient on x_{bkg} and pass through NN → Asimov bkg template
      i. Ideally remove gradient from already computed bkg shape, but depends on tensor library
7. Compute stats.:
   a. $t_{exp} = N_s \cdot shape_{sig} + N_b \cdot shape_{bkg}$
   b. $t_{asimov} = N_{s,true} \cdot shape_{sig} + N_{b,true} \cdot shape_{bkg,asimov}$

PROCESS - ONE UPDATE STEP

8. Build Poisson likelihood:
   a. NLL = -Pois(t_{exp}).log_prob(t_{asimov}).sum()
   b. N_s, N_b, and shape nuisances already at nominal values, NLL minimised, profiling unnecessary
   c. Add constraints on nuisances if present
9. Compute Hessian of NLL w.r.t. PoI and nuisances: ∇^2NLL (2D square matrix),
   a. N.B at minimum, ∇NLL = 0
   b. Hessian diagonal = effect of each param on NLL
   c. Hessian off-diagonal (symmetric) = interplay between params
10. Invert Hessian & return element corresponding to PoI as loss value
    a. Want PoI Hessian element to be as large as possible: NLL narrower in PoI axis
    b. But want nuisance elements to be as small as possible: NLL flatter in nuisance axes
    c. Inversion "mixes" elements in Hessian
    d. Minimising PoI element of inverted Hessian leads to desired result
11. Backprop loss value and update weights as normal

IMPLEMENTATION

Requirements

- Either:
  - Access to input data before forward pass (paper version)
  - Or access to pre-modified data for up/down systematic shifts (interpolation approximation version)
- Access to model when computing loss (might be avoidable in certain tensor libs)
  - Need to remove gradient due to nuisances on predictions

Difficulties

- Losses normally expected to be a function that receives only predictions and targets
- Most callbacks and recorders expect loss to be averaged over data in batch (i.e. non-reduced element-wises losses exist), but INFERNO is not an averaged quantity.

IMPLEMENTATION 1- CUSTOM FRAMEWORK

- Implement as a callback
  - Persistent class with access to the DNN
- on_batch_begin - modify data before forward pass
- on_forward_end - compute loss and manually set value
  - Requires training loop to be:
    - Aware of loss-setting-callbacks
    - Fine-grained enough (e.g. Keras 2 only has on_batch_begin on_batch_end (unsure about tf.keras))

```
def _fit_batch(self, x:Tensor, y:Tensor, w:Tensor) -> None:
    self.x,self.y,self.w = to_device(x,self.device),to_device(y,self.device),to_device(w,self.device)
    for c in self.cbs: c.on_batch_begin()
    self.y_pred = self.model(self.x)
    if self.state != 'test' and self.loss_func is not None:
        self.loss_func.weights = self.w
        self.loss_val = self.loss_func(self.y_pred, self.y)
    for c in self.cbs: c.on_forwards_end()
    if self.state != 'train': return

    self.opt.zero_grad()
    for c in self.cbs: c.on_backwards_begin()
    self.loss_val.backward()
    for c in self.cbs: c.on_backwards_end()
    self.opt.step()
    for c in self.cbs: c.on_batch_end()
```

Modify data to include nuisances

Compute loss and set self.loss_val

IMPLEMENTATION 2- EXISTING FRAMEWORK

- Implement as a callback
  - Persistent class with access to the DNN
  - Include __call__ method
  - Pass as both callback and loss function
- on_batch_begin - modify data before forward pass and stash a copy
- When called, compute & return loss
  - Divide loss by batch size so that it is correctly averaged
- *N.B. Right (Keras > 2.3) only used as example, might not actually work*

```
for step in data_handler.steps():
  with tf.profiler.experimental.Trace(
      'train',
      epoch_num=epoch,
      step_num=step,
      batch_size=batch_size,
      _r=1):
    callbacks.on_train_batch_begin(step)
    tmp_logs = self.train_function(iterator)
    if data_handler.should_sync:
      context.async_wait()
    logs = tmp_logs  # No error, now safe to assign to logs.
    end_step = step + data_handler.step_increment
    callbacks.on_train_batch_end(end_step, logs)
    if self.stop_training:
      break
```

Modify data to include nuisances

```
with tf.GradientTape() as tape:
  y_pred = self(x, training=True)
  loss = self.compiled_loss(
      y, y_pred, sample_weight, regularization_losses=self.losses)
self.optimizer.minimize(loss, self.trainable_variables, tape=tape)
self.compiled_metrics.update_state(y, y_pred, sample_weight)
return {m.name: m.result() for m in self.metrics}
```

Compute loss and set self.loss_val

EXISTING IMPLEMENTATIONS

- Tensorflow 1: paper-inferno - Pablo de Castro
  - Custom framework
  - Loss integrated into custom training loop
- Tensorflow 2: inferno - Lukas Layer
  - Single Jupyter Notebook (runnable on Colab)
  - Loss integrated into custom training loop
- PyTorch: pytorch_inferno - Me
  - Custom framework (pip installable)
  - Loss implemented as callback
  - LUMIN version foreseen

BLOG SERIES

- 5-part series on INFERNO and (param inference in HEP)
- Introduces & uses PyTorch package
- Parts 1 & 2 - intro to param inference
- Part 3 - ML classifier for summary stat.
- Part 4 - INFERNO for summary stat.
- Part 5 - Approximating INFERNO for easier application