Physics Dojo

Last Revised: A. Kit

February 19 2022

Integrate physics into the training cycle of DIVA

There are two methods: conservation between experimental and predicted quantities, and regularization of latent space through interpolation.

1 Conservation

Not only do we want a decent reconstruction of profiles, but that the decoded profiles maintain the same physical constraints as those that they are fitting. A physical constraint, by our definition, is a quantity, P, approximated from the experimental profile, that is then calculated in the same way from the predicted profile yielding the quantity, $\hat{P}$. We then try to get the model get these two quantities as close as possible through adding $\text{MSE}(P, \hat{P})$ to the loss function.

For example, the static electron pressure stored energy, $W_E \approx k_B \int p_e dV$, should be conserved through the encoding-decoding process. We can approximate this value by simply summing the **valid** (i.e., masked) pressure measurements, $W_e = \sum_{i=0}^{n} p_{e,i}$, (where $p_e = n_e T_e$ and $p_{e,i}$ is the pressure at a given measurement i along the spatial coordinates) and adding to the loss term $\text{MSE}(W_e, \hat{W}_e)$, where $\hat{W}_e$ is the decoded profile(s) static pressure.

We can do further approximations to determine α, the PB-instability limit. This is calculated in [FRASSINETI et al 2021 Nucl. Fus.] as

$$\alpha = -\frac{2\partial_\phi V}{(2\pi)^2}\left(\frac{V}{2\pi^2 R}\right)\mu_0 p'$$

, where V is the plasma volume enclosed by the flux surface, R is major radius, and p' is total pressure derivative in poloidal flux ϕ. This is normally calculated with ELITE [SOURCE]. As $\alpha := \alpha(\psi)$, we can choose the $\alpha_{crit} = \alpha(\psi = 1.0)$. We obtain V and $\partial_\psi V$, and R from EFIT, i.e., they are machine parameters. We then take $V(\psi = 1.0)$ (which is just the total volume of the plasma), $\partial V(\psi = 1.0)$ (the last value in VJAC from EFIT), and R for each time slice and plug them into our equation for α. However, we do not have $p'(\psi = 1.0)$, or rather we do, it is just not accurate, so we have to make an assumption that the maximum value of the pressure gradient profile is our $p'(\psi = 1.0)$, which we can then plug

into our equation above. Calculate for experimental profile and for decoded profile and add MSE($\alpha_{crit}, \hat{\alpha_{crit}}$) to the loss function.

2 Regularization

When moving throughout the latent space, the decoded profiles should still follow empirical 'rules'. One such rule is the greenwald density limit:

$$n_G = c\frac{I_P}{\pi a^2}$$

where I_P is the plasma current, a is the minor radius, and π has the ability to contain all information in the known world. The constant c is also empirically found so we need to mess around until we find one that is suitable.

We can move around the latent space per batch in a training epoch:

- Sample a handful of profiles and corresponding machine parameters from the batch:
  `in_prof_samp, in_mp_samp`
- Increase/decrease the current and or gas puff of samples
  `in_mp_samp -> in_mp_samp_varied`
- Pass varied samples through conditional prior reg into latent space
  `LS_sample_from_prior = COND_PRIOR_REG(in_mp_samp_varied)`
- Decode these latent space points
  `prof_from_prior = DECODER(LS_sample_from_prior)`
  `mp_from_prior = AUXREG(LS_sample_from_prior)`
- Check that profiles follow density limits by taking the MSE of density points w.r.t to density limit for points that land above/below density limits, the density limit is calculated using the **output** machine parameters.
  `MSE_from_density limit`
- Also take MSE between expected current and output of AUXREG since we used above outputs to calculate density limit.
  `MSE_expected_machine_parameters(in_mp_samp_varied, mp_from_prior)`
- Pass the encoded profile from prior into the encoder to get LS sample
  `LS_sample_from_enc = ENCODER(prof_from_prior)`
- Take MSE between the latent space samples (from encoder vs from prior)
  `MSE_enc_vs_prior = MSE(LS_sample_from_enc, LS_sample_from_prior)`
- Backprogopate the three MSEs.

Ideally this enforces two things: 1.) the latent space is regularized such that sampling from will return a profile that follows the density limit, and 2.) all components of the network learn this regularization since data is passed through all components.

2.1 Linear extrapolation method: WORK IN PROGRESS

The method of linear extrapolation within the latent space needs to be explored, but generally it could look something like this:

- Take highest/lowest current in batches
- Determine interpolation vector for given batch
- Extrapolate linearly beyond the vector
- For each point along the interpolation, check for the profile decoding with the Greenwald density limit (upper and lower limits).
- For each density point that does not follow the limit, calculate the MSE of it with , and back propogate.

3 Code

Assuming we are using PyTorch.

3.1 Conservation Rules

```
og_profs, mask, og_mps, _* = get_batch_data()
decoded_profs, decoded_mps = model.forward(og_profs)
# output profile and input profile
# They both have shape [BatchSize, 2, N],
# N: spatial resolution

# have to de_standardize to get to the real values
# it need not be cloned,
in_prof_ds = torch.clone(og_profs)
in_prof_ds[:, 0] = de_standardize(in_prof_ds[:, 0], D_mu, D_var)
in_prof_ds[:, 1] = de_standardize(in_prof_ds[:, 1], T_mu, T_var)

out_prof_ds = torch.clone(decoded_profs)
out_prof_ds[:, 0] = de_standardize(out_prof_ds[:, 0], D_mu, D_var)
out_prof_ds[:, 1] = de_standardize(out_prof_ds[:, 1], T_mu, T_var)

in_mps_ds = torch.clone(og_mps)
in_mps_ds = de_standardize(in_mps_ds, MP_mu, MP_var)

out_mps_ds = torch.clone(decoded_mps)
out_mps_ds = de_standardize(out_mps_ds, MP_mu, MP_var)


""" STORED ENERGY """
```

```
def calc_static_pressure_stored_energy(in_profs_ds, out_profs_ds, mask):

    boltz_con = 1.380e-23

    # torch.prod multiplies the density and temperature
    # to get pressure
    in_pres = torch.prod(in_ds.masked_fill_(~mask, 0), 1)
    out_pres = torch.prod(out_ds.masked_fill_(~mask, 0), 1)
    # ~inverse of mask since masked_fill_ uses True values,
    # Where our mask says False for bad vals.

    # .sum() them across the spatial dimension.
    stored_E_in = boltz_con*(in_pres).sum(1)
    stored_E_out = boltz_con*(out_pres).sum(1)
    return stored_E_in, stored_E_out

""" ALPHA CRITICAL """
def calc_alpha_crit(in_profs_ds, out_profs_ds,
                    in_mps_ds, out_mps_ds,
                    V_index=-2, VJAC_index=-1, R_index):

    mu_0 = 1.256e-6
    V_in, VJAC_in = in_mps_ds[:, V_index], in_mps_ds[:, VJAC_indx]
    R_in = in_mps_ds[:, R_index]

    V_out, VJAC_out = out_mps_ds[:, V_index], out_mps_ds[:, VJAC_indx]
    R_out = out_mps_ds[:, R_index]

    # Jury is out if we should use masks here
    in_pres = torch.prod(in_ds, 1)
    out_pres = torch.prod(out_ds, 1)

    in_pres_grad = torch.gradient(in_pres, 1)
    out_pres_prad = torch.gradient(out_pres, 1)

    in_pres_grad_min = torch.argmin(in_press_grad, dim=1, keepdim=True)
    out_pres_grad_min = torch.argmin(out_press_grad, dim=1, keepdim=True)
    in_pres_min = in_press[:, in_press_grad_min]
    out_pres_min = out_press[:, out_pres_grad_min]

    alpha_crit_in = -((2*V_JAC_in) / (2*torch.pi)**2)*
                    ((V_in / (2*R_in*torch.pi**2)**2) *
                    (mu_0 * in_press_min)
    alpha_crit_out = -((2*V_JAC_out) / (2*torch.pi)**2)*
                    ((V_out / (2*R_out*torch.pi**2)**2)*
                    (mu_0 * out_press_min)
```

```
    return alpha_crit_in, alpha_crit_out
```