Neural Networks and Debugging

Yash Belhe & Zhefan Xu

Recap

Now you know:

- What neural networks are and what they do (Lectures Week 1)
- How to train neural networks (Lectures Week 2 - 4)
- How do define models in PyTorch and how to optimize these models (Rec 1-3)

You have tried to use that knowledge in HW1P2.

It’s harder than recitations make you think.

Debugging Deep Learning

In Computer Science, debugging is always a big, painful part of the work. In Deep Learning it’s even bigger and more painful. Very often:

- You have implemented a sweet model
- The code looks fine
- Accuracy is terrible/you get a weird error
- Have no idea why?!

Debugging Deep Learning

The reason DL debugging is especially hard is that there’s a large number of phenomena that can make your code fail :

- Python-based error
- Pytorch-based error
- “Math” error (wrong minus somewhere)
- Modelization issue
- Training issue
- Testing issue
- ....

The hardest thing is to determine in which situation you are.

Plan for today

Today we'll cover these cases and what you can do about them (both **prevention** and **debugging**)

- General tips to **organize your code**
- Usual **Model-related errors** and how to find them
- Use metrics/hyperparams **visualization** to help you!

Apply these before coming to Office Hours!

General Coding Tips

Make your code modular

Design functions/classes for each of the main subtasks (data loading, model definition, model training,...).

Make sure you **thoroughly test** each one of these functions/ classes independently.

eg. once you define the dataset and the dataloader loop through the entire dataset to make sure that the data (both input data and targets) have the **correct shapes** that you expect

eg. make sure that the inputs and labels are **aligned** i.e does the input correspond to that output

Do not write your dataset, model and training loop without testing each one independently and expect everything to work.

General Coding Tips

Centralize your hyperparameters

Do **not** define/ hardcode the hyperparameters only in the file that they are being used in. This will make it **very hard** to iterate through different hyperparameter setups.

Instead, define all of the hyperparameters either in a separate **config file** that you pass to your main python script/ notebook, at a **single centralized location** within the main python script/ notebook or as **command line args** that you pass to the main python script

General Coding Tips

Start small

Use a simple model to start out, it will make debugging your code a lot easier. If this model works, you can slowly start increasing its complexity (num layers, num neurons, weight decay etc)

Make sure you can overfit to a single batch of data (more on this later)

Debugging

Coding error: when your model does not do what you want it to do (python, pytorch, math)

Training error: when your model does what you want it to do but is not learning well

Testing/decoding error: when your model is learning well but outputs bad results

(this one should be rare in HW1/HW2 but very usual when dealing with language in HW3/HW4. We won't talk too much about it.)

You should check for coding errors first, then training errors.

Coding Errors

Signs you may have one :

1. Loss does not decrease at all
2. Outputs are constants
3. Training stops mid-time for unclear reasons
4. Training takes FOREVER
5. Out of memory error

Coding Errors

How to find them : Print **everything** to look for the first moment the problem appears. Be methodical

What to check :

- Your **data** : Not iterating ? Instance-label misalignment ?
- Your **shapes** : everything consistent ?
- Your **hyperparameters** : when you print them, are they what they're supposed to be ?
- Your **parameters** : are they changing during training ? Are they going to 0 ? (they are in my_net.parameters())
- **Simpler cases** (ex : with batch size 1 does it work ?)
- Do this locally **before** you start training on AWS (Will save you a lot of money)

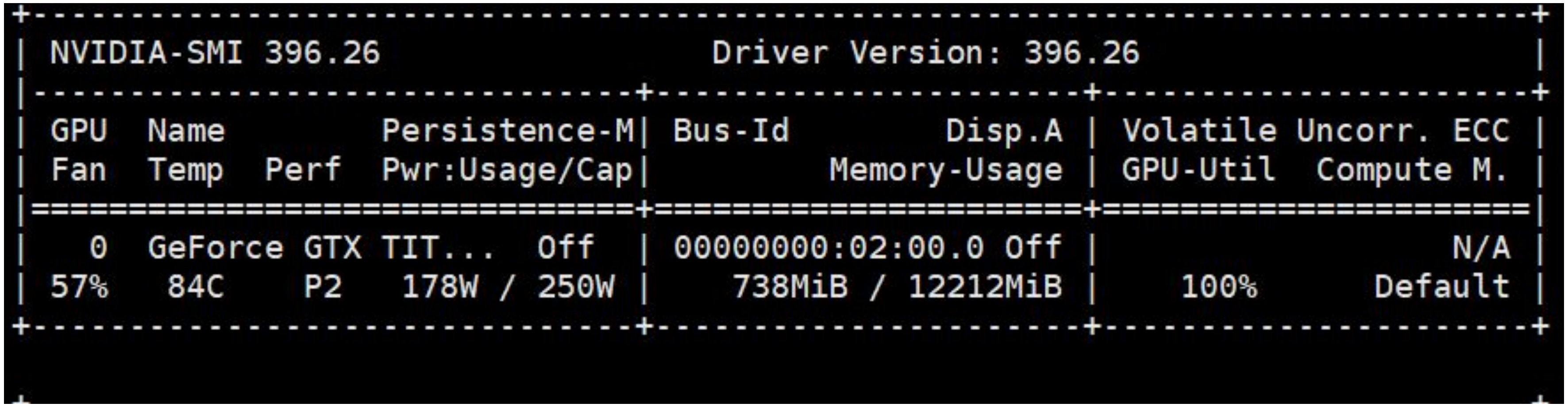
Time Issues

Specific case of coding error: when things work but are too slow.

In your epochs, use the **time** module to check the duration of all your subtasks (data loading, forward, backward,...), and find the aberrant one.

Make sure that your __getitem__ method is a **constant time** operation and does not iterate over the entire dataset

Make sure that your **model is on CUDA,** check this by running nvidia-smi



```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 396.26                 Driver Version: 396.26                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce GTX TIT...  Off  | 00000000:02:00.0 Off |                  N/A |
| 57%   84C    P2   178W / 250W |    738MiB / 12212MiB |    100%      Default |
+-------------------------------+----------------------+----------------------+
```

Time Issues

If your model is on CUDA and your GPU utilization is low, the issue is probably in your dataloader, double check it to make sure that the __getitem__ function is constant time

Forcing your model, data and targets to cuda using .cuda() will ensure that you’re on CUDA or throw an error if it isn’t available

Training Errors

For these errors, usually your loss does decrease, but not enough. If you see absurdly low performance (~random) it's probably a coding error.

Note : a random classification model would have a cross-entropy loss of ~ln(Number_of_classes). Why?

Training Errors

Different problems:

Modelization issues: your model is too small to learn patterns (or not well designed when the problem is complex)

Optimization issues: you cannot train your model properly

Overfitting: your model is too big/you train too long

Modelization/ Optimization Issues

Sign that you have one : the **training** loss does not go down well enough

You know that:

A good model will overfit if you train it too long → you can use that to debug

Train your model on a small subset of your data for many epochs (i.e **overfit your model** on that data): the training loss should go to 0. If it doesn't, you have a problem.

Modelization issues

Is your model too simple/ weak?

You should refer to literature/your experience to know that. If you feel that this is the issue, you can easily fix it by simply increasing the number of layers/ number of neurons per layer

If you're sure that you don't have a modelization issue, look for an optimization issue.

Optimization issues

You should look for :

● Learning rate: if too small you will learn too slowly. If too large you will learn for a while then diverge.

Default “good” for ADAM: 0.001.

It is recommended to do learning rate decay : start large, then decrease (for example when loss stops improving) See PyTorch docs for different LR Decay strategies (ReduceLROnPlateau, StepLR, etc.)

● Optimizer (default “good” : Adam)

● Initialization (default “good” : xavier)

● Batching (just the batch size on simple problems). Default “good” : from 32 to 128 if you can afford it. Very large batch sizes may be troublesome if you don’t scale the LR appropriately too

Too deep models can create optimization problems too (vanishing gradients). They also lead to...

Overfitting issues

How do you know whether you're overfitting or not? **Training loss decreases but validation loss doesn't.**

You should **always** have a small validation set to look at every epoch. You should **always** log/ plot your training and validation losses at each epoch.

Things to do there :

- Verify that you shuffle your training data
- Decrease your model size/depth
- Use some of the tricks to **increase regularization**:

dropout, batchnorm, early stopping, weight decay etc.

Note : adaptative optimizers (Adam,...) overfit more.

Overfitting issues

It’s also possible to overfit on the validation set.

This happens when you try a very large amount of architectures/hyperparameters with the same validation set: you may find one that works “by chance” and won’t generalize.

(In HW1P2 : if you do 200 attempts a day on kaggle, you may overfit on the public leaderboard and be disappointed by your results on the private leaderboard).

Testing/decoding issues

When your model learns, training and validation loss decrease, but accuracy is low.

Recall that losses (ex:cross-entropy) are differentiable surrogates for the metric you want (ex:accuracy). It’s always possible to have a gap between the two.

On a simple classification problem like HW1P2 this shouldn’t happen too much (unless bug in the prediction function). However, to be safe you should look at your validation accuracy along with your loss.

Visualize your metrics

We repeated many times that you should look at your metrics, compare training/validation loss, accuracy etc.

But, just printing them in the terminal is dirty and hard to read. That’s why you should visualize them

→ Second part of this recitation

Why Visualize?

- Answers the question “What am I learning?”
- To see how your weight matrix and gradients change over time during training of your model, which can help determine whether you need to:
  - Remove extra layers when there is a redundancy in them
  - Add new layers to see if they learn something unique
- To predict the right time to stop training the model
  - It’s better to use tools to predict when to stop rather than logging loss and accuracies at each step of training
- Weight Initialization
  - To better understand which weight initialization method performs better for the given problem
  - We get to see why initializing with zeroes is not preferred
- To compare validation vs training accuracy - you can easily spot overfitting
- To compare accuracies across different runs to see the effect of different hyperparameters

Why Visualize?

How well are the Activation functions performing?

Is the Dropout rate too high?

In general, visualization helps to fine tune the network for better or optimal performance

Tensorboard

TensorBoard is a visualization library for TensorFlow that is useful in understanding training runs, tensors, and graphs.

You can plot time series data for scalars such as training/validation loss and accuracy over time

You can plot the gradients to inspect whether you have a problem of vanishing or exploding gradients

Starting Tensorboard

Forward the remote 6006 port to your local 6007 port:

```
ssh -i KeyTest.pem -L 8000:localhost:8888 -L
6007:localhost:6006
```

To Install: `pip install tensorboard`

To start Tensorboard, run the following command from your terminal in AWS (don't forget to activate your pytorch_p36 virtual environment first):

```
tensorboard --logdir=./runs
```

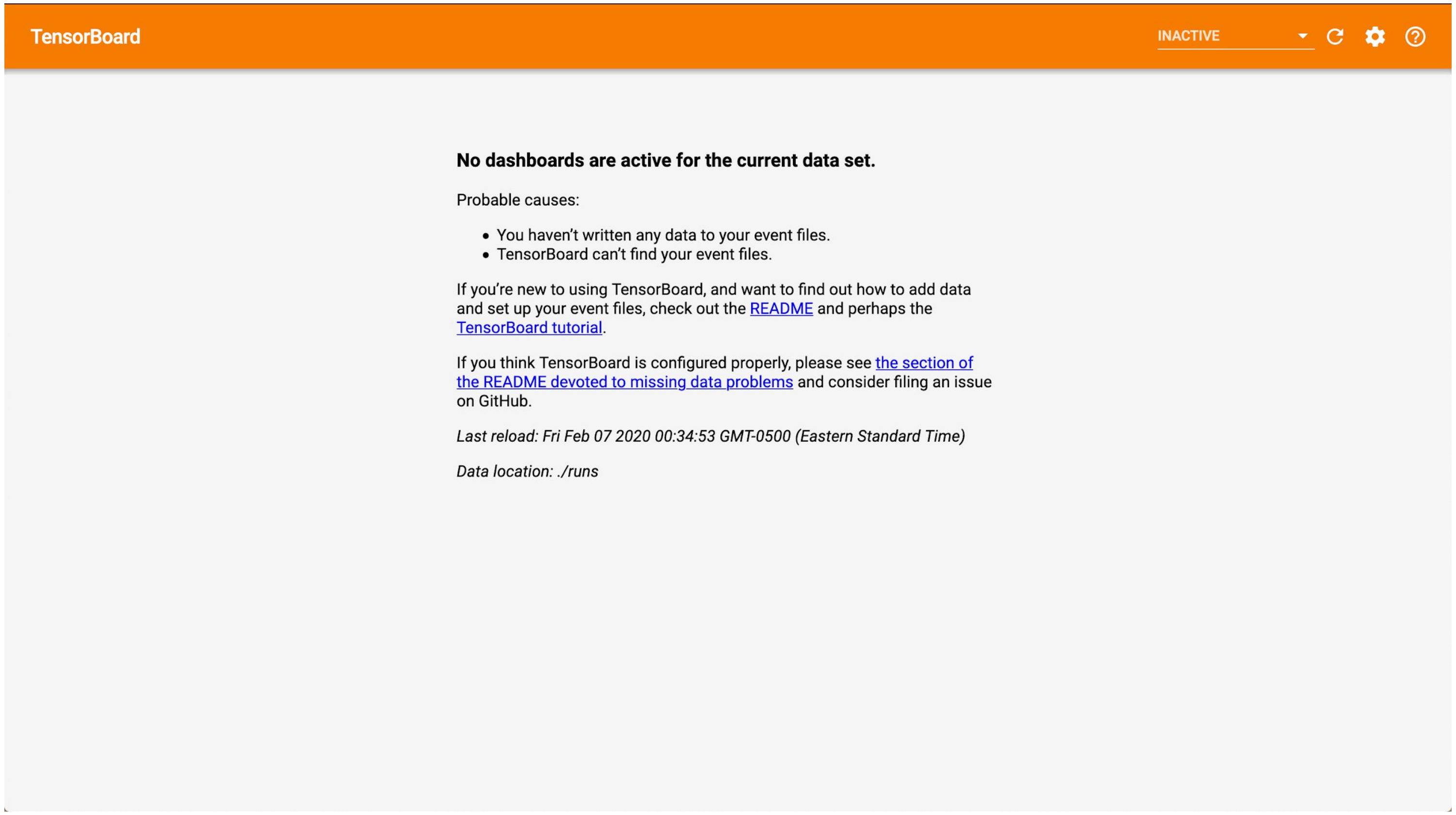
Where ./runs is the directory for Tensorboard to search for the event files.

Next, type the following link in your local browser to view Tensorboard's main dashboard:

```
localhost:6007
```

Starting Tensorboard

If you correctly follow the instructions in the previous two slides, you should be seeing something like this in your browser:



No dashboards are active for the current data set.

Probable causes:

- You haven't written any data to your event files.
- TensorBoard can't find your event files.

If you're new to using TensorBoard, and want to find out how to add data and set up your event files, check out the README and perhaps the TensorBoard tutorial.

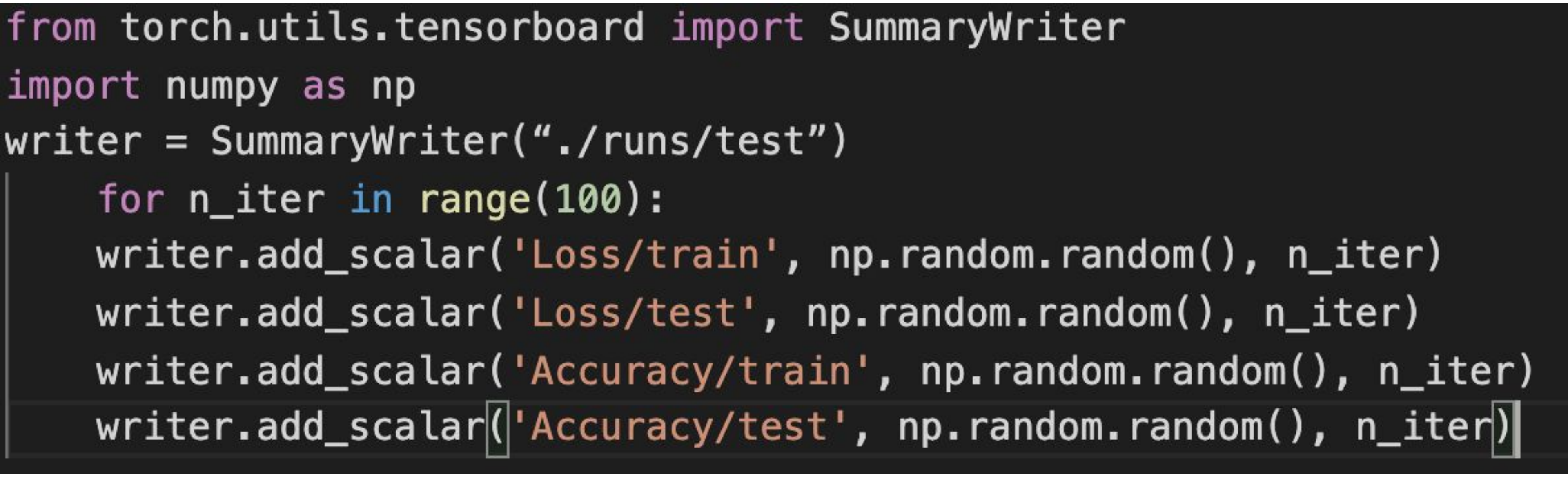
If you think TensorBoard is configured properly, please see the section of the README devoted to missing data problems and consider filing an issue on GitHub.

Last reload: Fri Feb 07 2020 00:34:53 GMT-0500 (Eastern Standard Time)

Data location: ./runs

Testing Tensorboard

Run the following python script to log random data to the directory you specified when you started tensorboard:



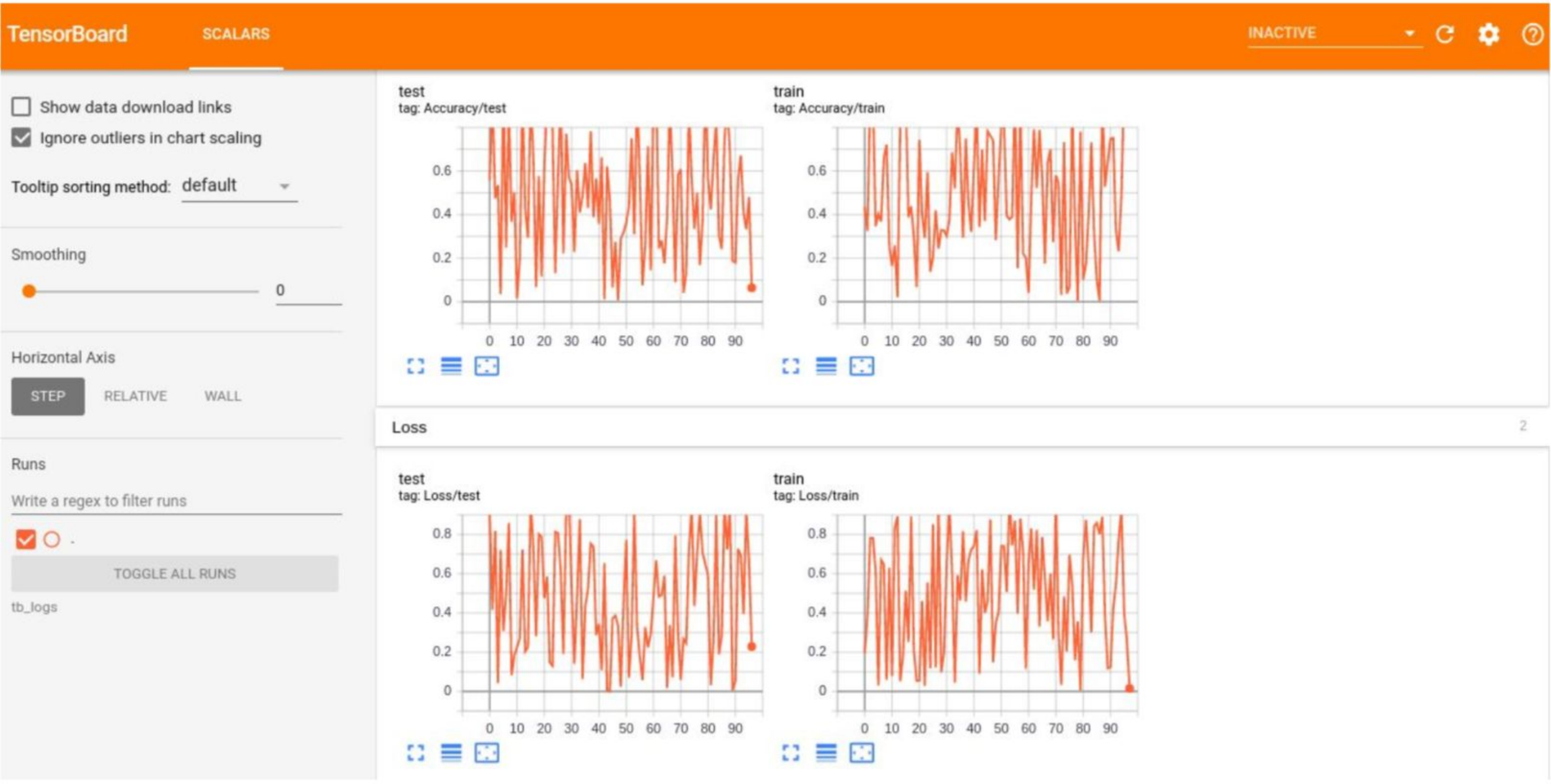
```
from torch.utils.tensorboard import SummaryWriter
import numpy as np
writer = SummaryWriter(“./runs/test”)
    for n_iter in range(100):
    writer.add_scalar('Loss/train', np.random.random(), n_iter)
    writer.add_scalar('Loss/test', np.random.random(), n_iter)
    writer.add_scalar('Accuracy/train', np.random.random(), n_iter)
    writer.add_scalar('Accuracy/test', np.random.random(), n_iter)
```

SummaryWriter(‘./runs_directory/current_experiment_name’): The object that you will use to send all of your data to be graphed to

add_scalar(‘Variable_group/Variable_name’, Variable_value, current_index): Used to plot scalars such as accuracy and error

Testing Tensorboard

If everything works well, you should see this:



Create SummaryWriter

A SummaryWriter writes all values we want to visualize to event files in a given directory.

You should use different run directories (“example”) in a common root directory (“./runs”) for different runs of your model. Ideally you should replace example with something more descriptive such as “exp_278_lr_0001_wd_1e5_bn_false...” OR maintain a log of exactly what hyperparameters you used for each experiment.

Will also help you easily compare the performance of different model/ hyperparameter configurations

```
from torch.utils.tensorboard import SummaryWriter
writer = SummaryWriter("./runs/example")
```

Add Values to Tensorboard

All add_ methods of SummaryWriter take 3 parameters: tag, value, step.

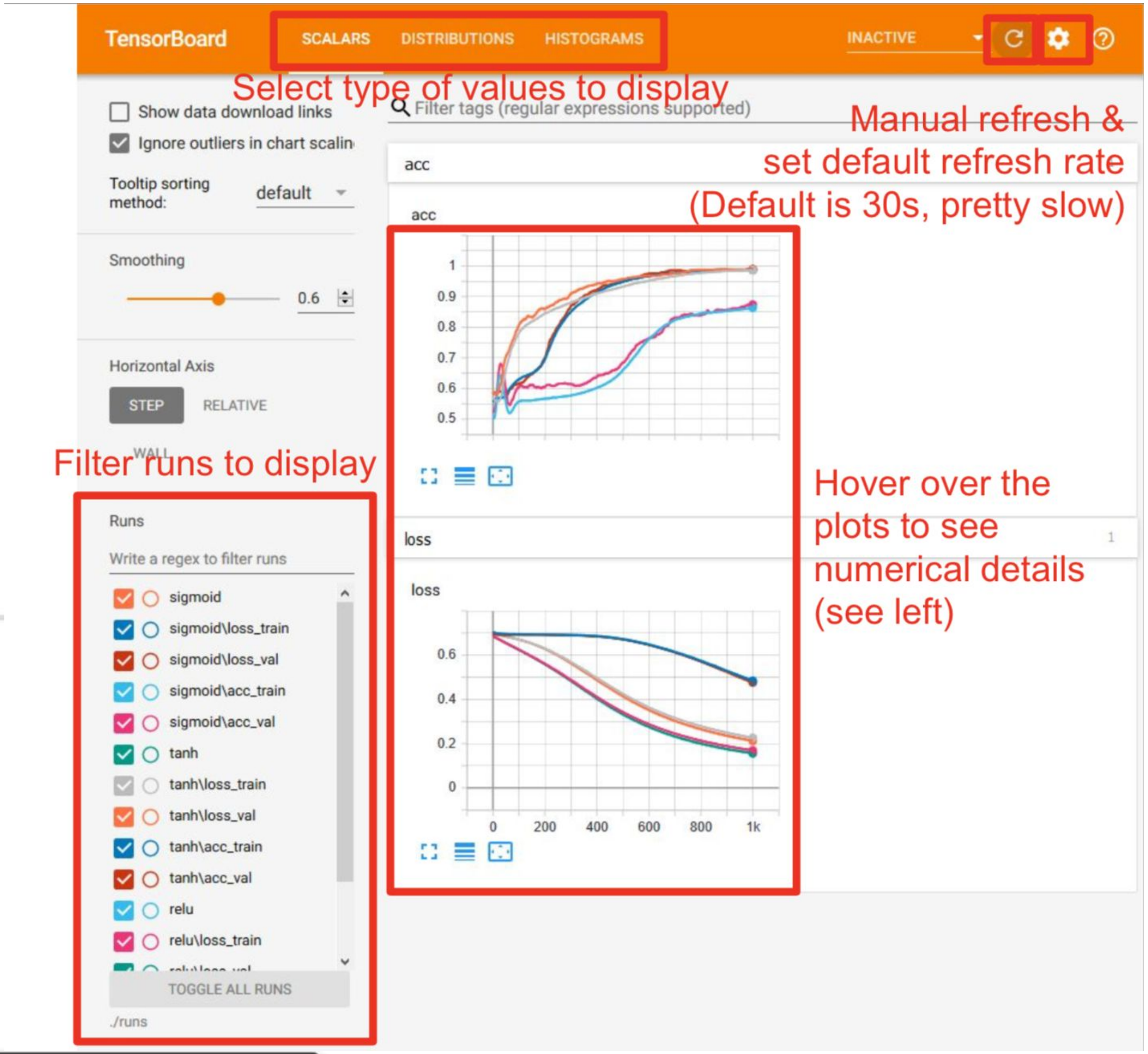
Step is an increasing integer, usually the number of iterations, that serves as the x-axis value of the plot.

Each method appends a data point to a type of plot. Common methods are:

- add_scalar: line plot of one variable (value is a scalar, i.e. a float number).
- add_scalars: line plot of multiple variables (value is a dict of scalars).
- add_histogram: histogram of distributions of values (value is a tensor). Useful for understanding the dynamics of the network.

You can even add fancier values, like “add_image”, “add_audio”.

TensorBoard UI



Select type of values to display

Manual refresh & set default refresh rate (Default is 30s, pretty slow)

Filter runs to display

Hover over the plots to see numerical details (see left)

☐ Show data download links

☑ Ignore outliers in chart scalin

Tooltip sorting method: default

Smoothing: 0.6

Horizontal Axis: [✓ STEP] RELATIVE WALL

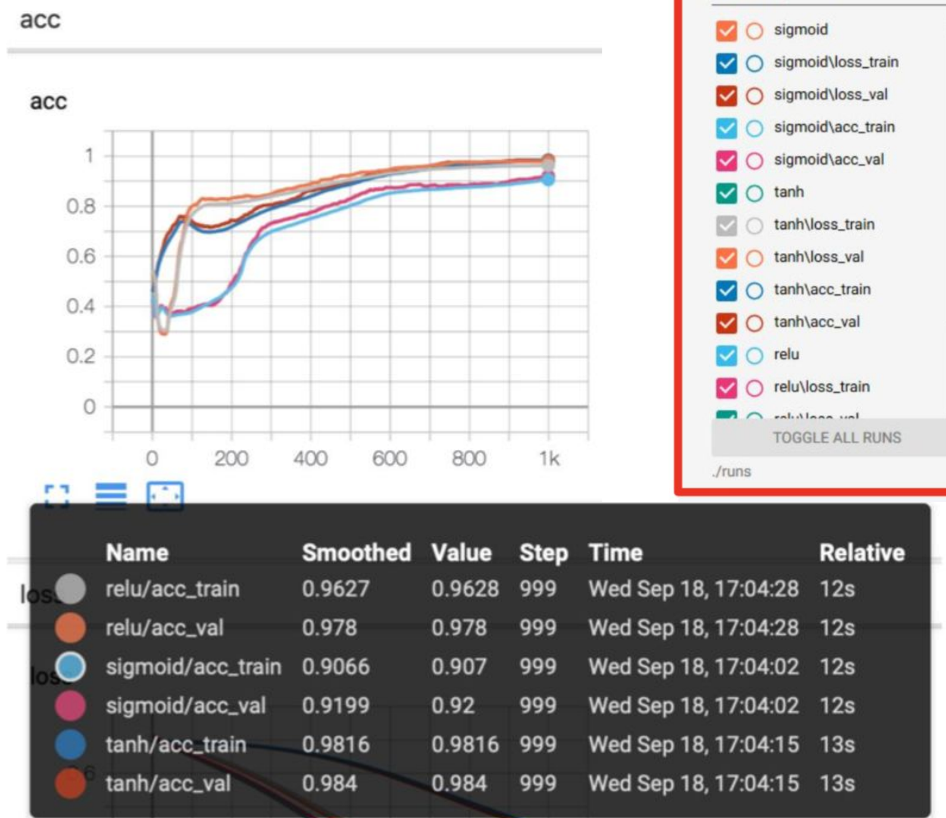
Filter tags (regular expressions supported)

Runs

Write a regex to filter runs

- ☑ sigmoid
- ☑ sigmoid\loss_train
- ☑ sigmoid\loss_val
- ☑ sigmoid\acc_train
- ☑ sigmoid\acc_val
- ☑ tanh
- ☐ tanh\loss_train
- ☑ tanh\loss_val
- ☑ tanh\acc_train
- ☑ tanh\acc_val
- ☑ relu
- ☑ relu\loss_train

./runs

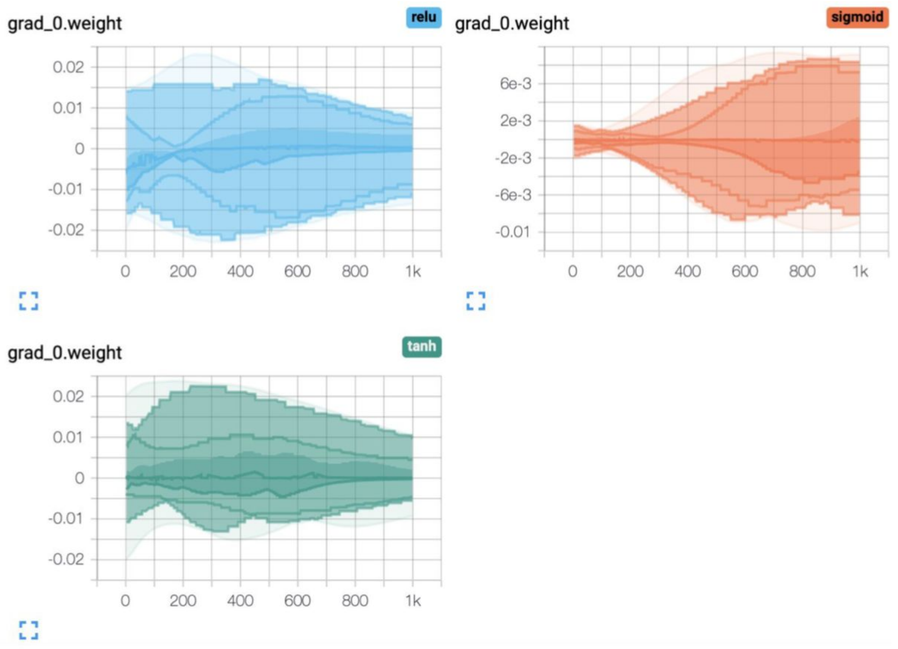


	Name	Smoothed	Value	Step	Time	Relative
●	relu/acc_train	0.9627	0.9628	999	Wed Sep 18, 17:04:28	12s
●	relu/acc_val	0.978	0.978	999	Wed Sep 18, 17:04:28	12s
●	sigmoid/acc_train	0.9066	0.907	999	Wed Sep 18, 17:04:02	12s
●	sigmoid/acc_val	0.9199	0.92	999	Wed Sep 18, 17:04:02	12s
●	tanh/acc_train	0.9816	0.9816	999	Wed Sep 18, 17:04:15	13s
●	tanh/acc_val	0.984	0.984	999	Wed Sep 18, 17:04:15	13s

Understanding Histograms

Plot histograms of gradients of network parameters.

The gradient vanishing issue with sigmoid activation is obvious on the histogram (orange).


grad_0.weight
relu
1k
grad_0.weight
sigmoid
6e-3
2e-3
-2e-3
-6e-3
1k
grad_0.weight
tanh
1k