Misc tools and hyperparameter tuning

Samuel Cheng

Some key components of training ML models

- Data preparation
- Model architecture and loss function design
- Optimization choice
- Training loop
  - Iterating over training data, feeding it into model, calculating loss, and update parameters by optimizers and hyperparameters through schedulers
- Evaluation
- Hyperparameter tuning

Overview of PyTorch Lightning

- The goal of PyTorch Lightning is to simplify and standardize the training process, while still providing full access to the power and flexibility of PyTorch.
- Key features:
  - Reproducibility: PyTorch Lightning provides a standardized training loop and a set of best practices to ensure that models can be trained consistently across different machines and environments.
  - Code readability: PyTorch Lightning separates the boilerplate code for training and validation from the model architecture, making the code more readable and easier to debug.
  - Scalability: PyTorch Lightning provides a simple and efficient way to distribute training across multiple GPUs or machines.
  - Flexibility: PyTorch Lightning allows you to customize the training process to suit your needs, while still providing a standardized interface for common tasks.
  - Community-driven development: PyTorch Lightning is an open-source project that is actively maintained and developed by a growing community of users and contributors.

Review of terminologies

- Dataset: training/validation/test (70/15/15)
- Dataloader: load a batch at a time
- Step vs epoch: each step load a new batch, each epoch go through all training data

Torch.utils.data.Dataset

- Require functions:
  - __init__(self,…)
  - __len__(self)
    - Return number of samples in dataset
  - __getitem__(self, i)
    - Return the the data and label of the I-th sample

```python
import torch
from torch.utils.data import Dataset

class MyDataset(Dataset):
    def __init__(self, data, labels):
        self.data = data
        self.labels = labels

    def __len__(self):
        return len(self.data)

    def __getitem__(self, index):
        x = self.data[index]
        y = self.labels[index]
        return x, y
```

Torch.util.data.DataLoader

Let's use simple 1D regression problem as an example

```
from torch.utils.data import DataLoader

N=10000
x = torch.unsqueeze(torch.linspace(-1, 1, N), dim=1)
y = x.pow(2) + 0.2*torch.rand(x.size())

my_dataset = MyDataset(x,y)
my_dataloader = DataLoader(my_dataset, batch_size=100, shuffle=True)
```

Barebone lightning module

```
import pytorch_lightning as pl
import torch.nn.functional as F
import torch

class myLightningModule(pl.LightningModule):
    def __init__(self, n_hidden):
        super().__init__()
        self.net = Net(n_feature=1, n_hidden=n_hidden, n_output=1)

    def training_step(self, batch, batch_idx):
        # training_step defines the train loop.
        x, y = batch
        y_hat = self.net(x)
        loss = F.mse_loss(y_hat, x)
        return loss

    def configure_optimizers(self):
        optimizer = torch.optim.SGD(self.parameters(), lr=0.2)
        return optimizer

# net = Net (n_feature=1, n_hidden=10, n_output =1)
trainer = pl.Trainer(accelerator='gpu', devices=1, max_epochs=10, default_root_dir="./lightning-example")
lightmodule = myLightningModule(n_hidden=10)

trainer.fit(model=lightmodule, train_dataloaders=my_dataloader)
```

Barebone lightning module

```
import pytorch_lightning as pl
import torch.nn.functional as F
import torch

class myLightningModule(pl.LightningModule):
    def __init__(self, n_hidden):
        super().__init__()
        self.net = Net(n_feature=1, n_hidden=n_hidden, n_output=1)

    def training_step(self, batch, batch_idx):
        # training_step defines the train loop.
        x, y = batch
        y_hat = self.net(x)
        loss = F.mse_loss(y_hat, x)
        return loss

    def configure_optimizers(self):
        optimizer = torch.optim.SGD(self.parameters(), lr=0.2)
        return optimizer

# net = Net (n_feature=1, n_hidden=10, n_output =1)
trainer = pl.Trainer(accelerator='gpu', devices=1, max_epochs=10, default_root_dir="./lightning-example")
lightmodule = myLightningModule(n_hidden=10)

trainer.fit(model=lightmodule, train_dataloaders=my_dataloader)
```

Barebone lightning module

```python
import pytorch_lightning as pl
import torch.nn.functional as F
import torch

class myLightningModule(pl.LightningModule):
    def __init__(self, n_hidden):
        super().__init__()
        self.net = Net(n_feature=1, n_hidden=n_hidden, n_output=1)

    def training_step(self, batch, batch_idx):
        # training_step defines the train loop.
        x, y = batch
        y_hat = self.net(x)
        loss = F.mse_loss(y_hat, x)
        return loss

    def configure_optimizers(self):
        optimizer = torch.optim.SGD(self.parameters(), lr=0.2)
        return optimizer

# net = Net (n_feature=1, n_hidden=10, n_output =1)
trainer = pl.Trainer(accelerator='gpu', devices=1, max_epochs=10, default_root_dir="./lightning-example")
lightmodule = myLightningModule(n_hidden=10)

trainer.fit(model=lightmodule, train_dataloaders=my_dataloader)
```

Barebone lightning module

```
import pytorch_lightning as pl
import torch.nn.functional as F
import torch

class myLightningModule(pl.LightningModule):
    def __init__(self, n_hidden):
        super().__init__()
        self.net = Net(n_feature=1, n_hidden=n_hidden, n_output=1)

    def training_step(self, batch, batch_idx):
        # training_step defines the train loop.
        x, y = batch
        y_hat = self.net(x)
        loss = F.mse_loss(y_hat, x)
        return loss

    def configure_optimizers(self):
        optimizer = torch.optim.SGD(self.parameters(), lr=0.2)
        return optimizer

# net = Net (n_feature=1, n_hidden=10, n_output =1)
trainer = pl.Trainer(accelerator='gpu', devices=1, max_epochs=10, default_root_dir="./lightning-example")
lightmodule = myLightningModule(n_hidden=10)

trainer.fit(model=lightmodule, train_dataloaders=my_dataloader)
```

Barebone lightning module

```python
import pytorch_lightning as pl
import torch.nn.functional as F
import torch

class myLightningModule(pl.LightningModule):
    def __init__(self, n_hidden):
        super().__init__()
        self.net = Net(n_feature=1, n_hidden=n_hidden, n_output=1)

    def training_step(self, batch, batch_idx):
        # training_step defines the train loop.
        x, y = batch
        y_hat = self.net(x)
        loss = F.mse_loss(y_hat, x)
        return loss

    def configure_optimizers(self):
        optimizer = torch.optim.SGD(self.parameters(), lr=0.2)
        return optimizer

# net = Net (n_feature=1, n_hidden=10, n_output =1)
trainer = pl.Trainer(accelerator='gpu', devices=1, max_epochs=10, default_root_dir="./lightning-example")
lightmodule = myLightningModule(n_hidden=10)

trainer.fit(model=lightmodule, train_dataloaders=my_dataloader)
```

Barebone lightning module

```python
import pytorch_lightning as pl
import torch.nn.functional as F
import torch

class myLightningModule(pl.LightningModule):
    def __init__(self, n_hidden):
        super().__init__()
        self.net = Net(n_feature=1, n_hidden=n_hidden, n_output=1)

    def training_step(self, batch, batch_idx):
        # training_step defines the train loop.
        x, y = batch
        y_hat = self.net(x)
        loss = F.mse_loss(y_hat, x)
        return loss

    def configure_optimizers(self):
        optimizer = torch.optim.SGD(self.parameters(), lr=0.2)
        return optimizer

# net = Net (n_feature=1, n_hidden=10, n_output =1)
trainer = pl.Trainer(accelerator='gpu', devices=1, max_epochs=10, default_root_dir="./lightning-example")
lightmodule = myLightningModule(n_hidden=10)

trainer.fit(model=lightmodule, train_dataloaders=my_dataloader)
```

Barebone lightning module

```
import pytorch_lightning as pl
import torch.nn.functional as F
import torch

class myLightningModule(pl.LightningModule):
    def __init__(self, n_hidden):
        super().__init__()
        self.net = Net(n_feature=1, n_hidden=n_hidden, n_output=1)

    def training_step(self, batch, batch_idx):
        # training_step defines the train loop.
        x, y = batch
        y_hat = self.net(x)
        loss = F.mse_loss(y_hat, x)
        return loss

    def configure_optimizers(self):
        optimizer = torch.optim.SGD(self.parameters(), lr=0.2)
        return optimizer

# net = Net (n_feature=1, n_hidden=10, n_output =1)
trainer = pl.Trainer(accelerator='gpu', devices=1, max_epochs=10, default_root_dir="./lightning-example")
lightmodule = myLightningModule(n_hidden=10)

trainer.fit(model=lightmodule, train_dataloaders=my_dataloader)
```

Adding validation and test steps

```python
def validation_step(self, batch, batch_idx):
    x, y = batch
    y_hat = self.net(x)
    loss = F.mse_loss(y_hat, x)
    return loss

def test_step(self, batch, batch_idx):
    x, y = batch
    y_hat = self.net(x)
    loss = F.mse_loss(y_hat, x)
    return loss
```

```python
import torch.utils.data as data
train_set_size = int(len(my_dataset) * 0.7)
test_set_size = int(len(my_dataset) * 0.15)
valid_set_size = len(my_dataset) - train_set_size - test_set_size

train_set, valid_set, test_set = data.random_split(my_dataset,
        [train_set_size, valid_set_size, test_set_size], generator=torch.Generator().manual_seed(42))
train_loader = DataLoader(train_set, batch_size=100, shuffle=True)
valid_loader = DataLoader(valid_set, batch_size=100, shuffle=True)

trainer = pl.Trainer(accelerator='gpu', devices=1, max_epochs=10) #callbacks=[lr_monitor_callback],logger=logger)
lightmodule = myLightningModule(n_hidden=10)
trainer.fit(lightmodule, train_loader, valid_loader)
```

Log average validation error with tensorboard

```
def validation_step(self, batch, batch_idx):
    x, y = batch
    y_hat = self.net(x)
    loss = F.mse_loss(y_hat, x)
    return {'val_loss': loss}

def validation_epoch_end(self, outputs):
    avg_loss = torch.stack([x['val_loss'] for x in outputs]).mean()
    self.log('validation_loss',avg_loss)
    log = {'val_loss':avg_loss}
```

```
# train with both splits
logger = TensorBoardLogger('tb_logs',name='my_model')
trainer = pl.Trainer(accelerator='gpu', devices=1, max_epochs=10,logger=logger)
lightmodule = myLightningModule(n_hidden=10)
trainer.fit(lightmodule, train_loader, valid_loader)
```

```
from pytorch_lightning.loggers import TensorBoardLogger
```

Log average validation error with tensorboard

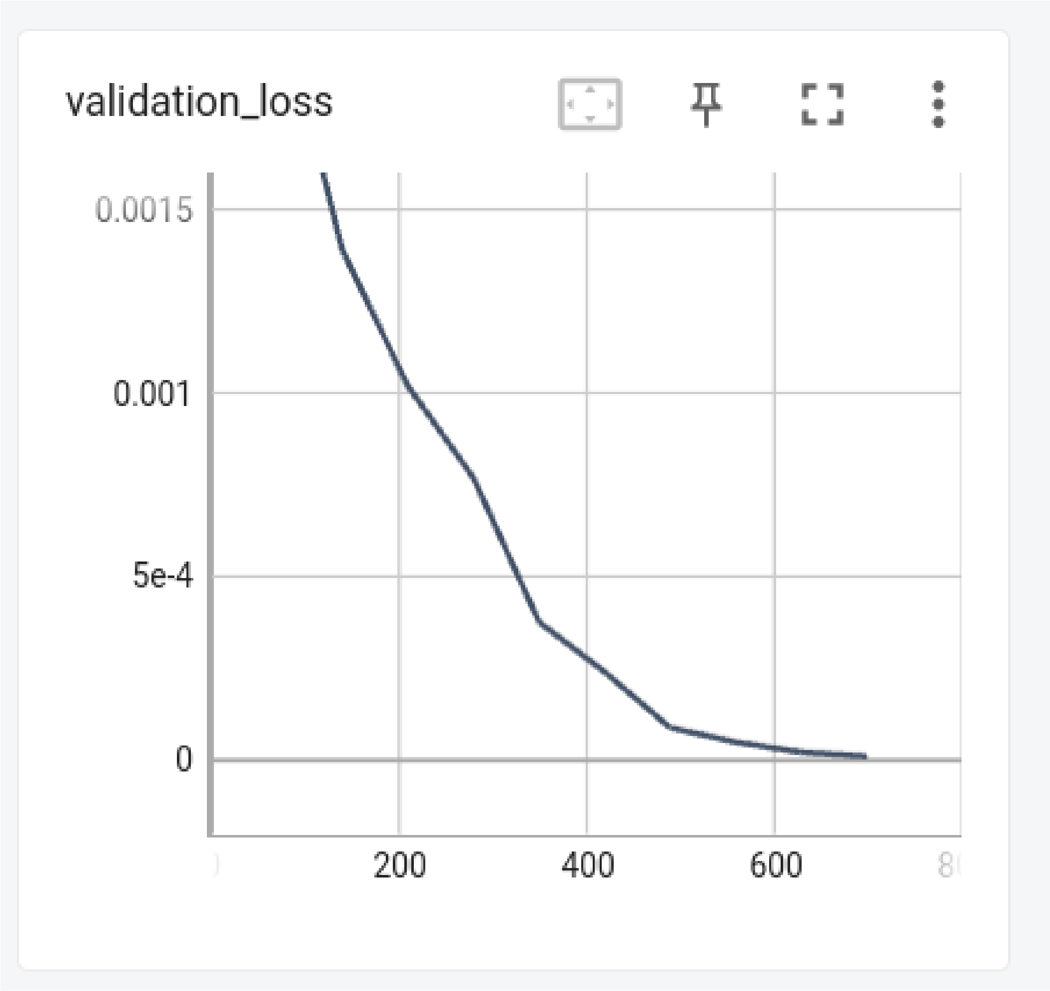
```python
def validation_step(self, batch, batch_idx):
    x, y = batch
    y_hat = self.net(x)
    loss = F.mse_loss(y_hat, x)
    return {'val_loss': loss}

def validation_epoch_end(self, outputs):
    avg_loss = torch.stack([x['val_loss'] for x in outputs]).mean()
    self.log('validation_loss',avg_loss)
    log = {'val_loss':avg_loss}
```

```python
# train with both splits
logger = TensorBoardLogger('tb_logs',name='my_model')
trainer = pl.Trainer(accelerator='gpu', devices=1, max_epochs=10,logger=logger)
lightmodule = myLightningModule(n_hidden=10)
trainer.fit(lightmodule, train_loader, valid_loader)
```

```python
from pytorch_lightning.loggers import TensorBoardLogger
```

tensorboard --logdir tb_logs --port 6006



Turn off stuffs you don't need

```
def lightning_loop(cls_model, idx, device_type: str = "cuda", num_epochs=10):
    seed_everything(idx)
    torch.backends.cudnn.deterministic = True

    model = cls_model()
    # init model parts
    trainer = Trainer(
        # as the first run is skipped, no need to run it long
        max_epochs=num_epochs if idx > 0 else 1,
        enable_progress_bar=False,
        enable_model_summary=False,
        enable_checkpointing=False,
        gpus=1 if device_type == "cuda" else 0,
        logger=False,
        replace_sampler_ddp=False,
    )
    trainer.fit(model)

    return trainer.fit_loop.running_loss.last().item(), _hook_memory()
```

TURN THESE OFF (AS SHOWN)!

Get a summary of a model with torchsummary

In [95]:
```
from torchsummary import summary

net=Net(1,10,1)
summary(net, input_size=tuple([1]), device='cpu')
```

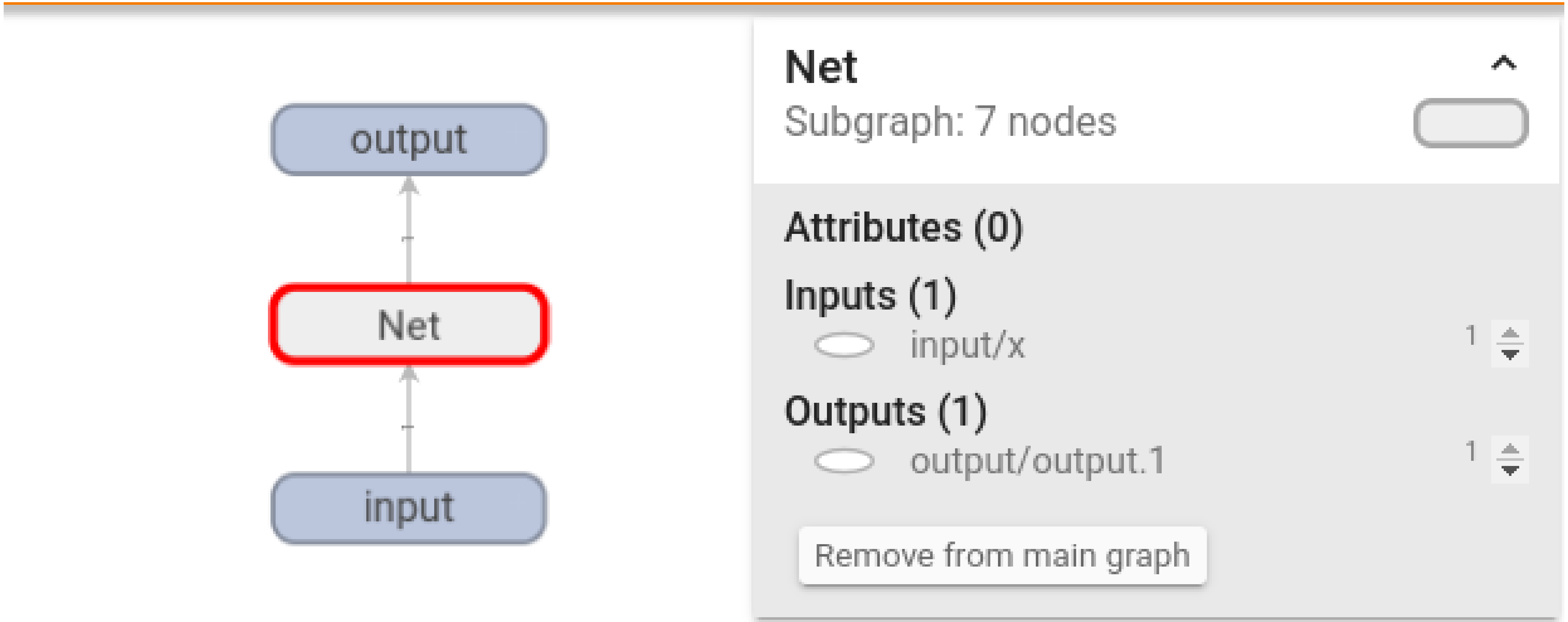
```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1                   [-1, 10]              20
            Linear-2                    [-1, 1]              11
================================================================
Total params: 31
Trainable params: 31
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.00
Params size (MB): 0.00
Estimated Total Size (MB): 0.00
----------------------------------------------------------------
```

Get a summary of a model for tensorboard

```
from torch.utils.tensorboard import SummaryWriter

writer = SummaryWriter()
net=Net(1,10,1)
inputs = torch.randn(1)
writer.add_graph(net, inputs)
```

tensorboard --logdir runs --port 6006

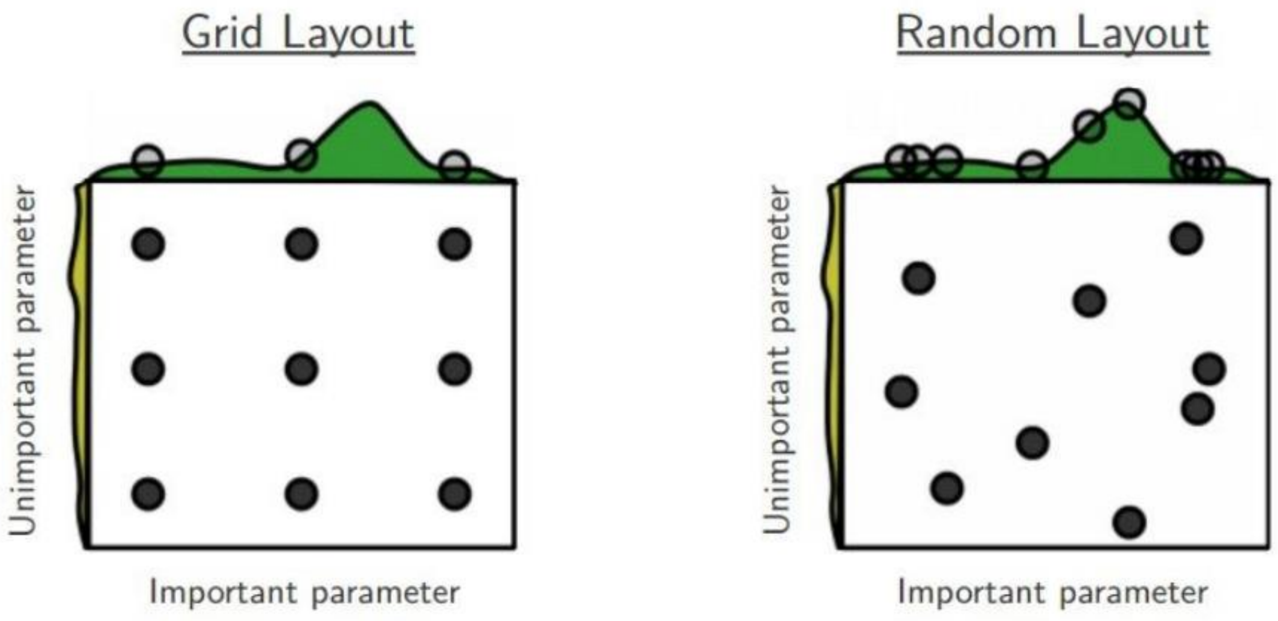


Some training advice from Andrej Karpathy

- Double check if loss is reasonable
    - E.g., can crank up regularization and training loss should increase
- Double check if model is reasonable. With little or no regularization,
    - Should be able to overfit a small training dataset . i.e., training error = 0
- Check learning rate is too small or too large
    - Too small: barely learning anything
    - Too large: NaNs

Never use grid search

- Hyperparameters: LR, # layers, # neurons in a layer, optimizers, etc.
- Avoid grid search in particular if you have many hyperparameters

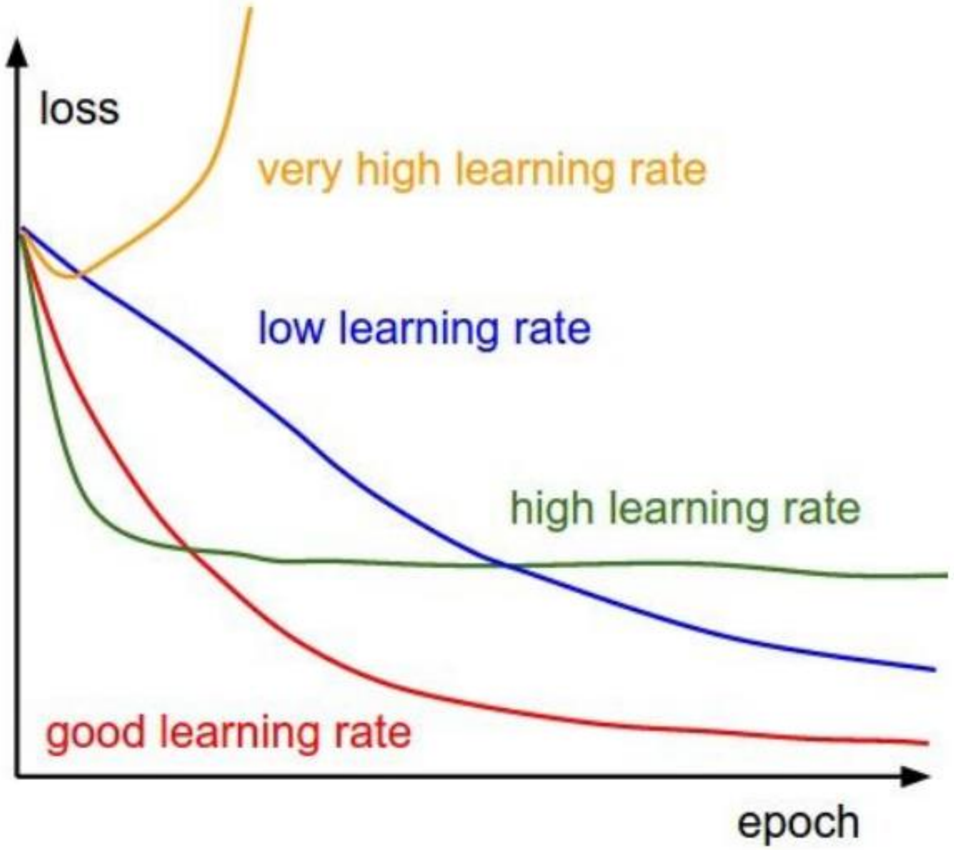


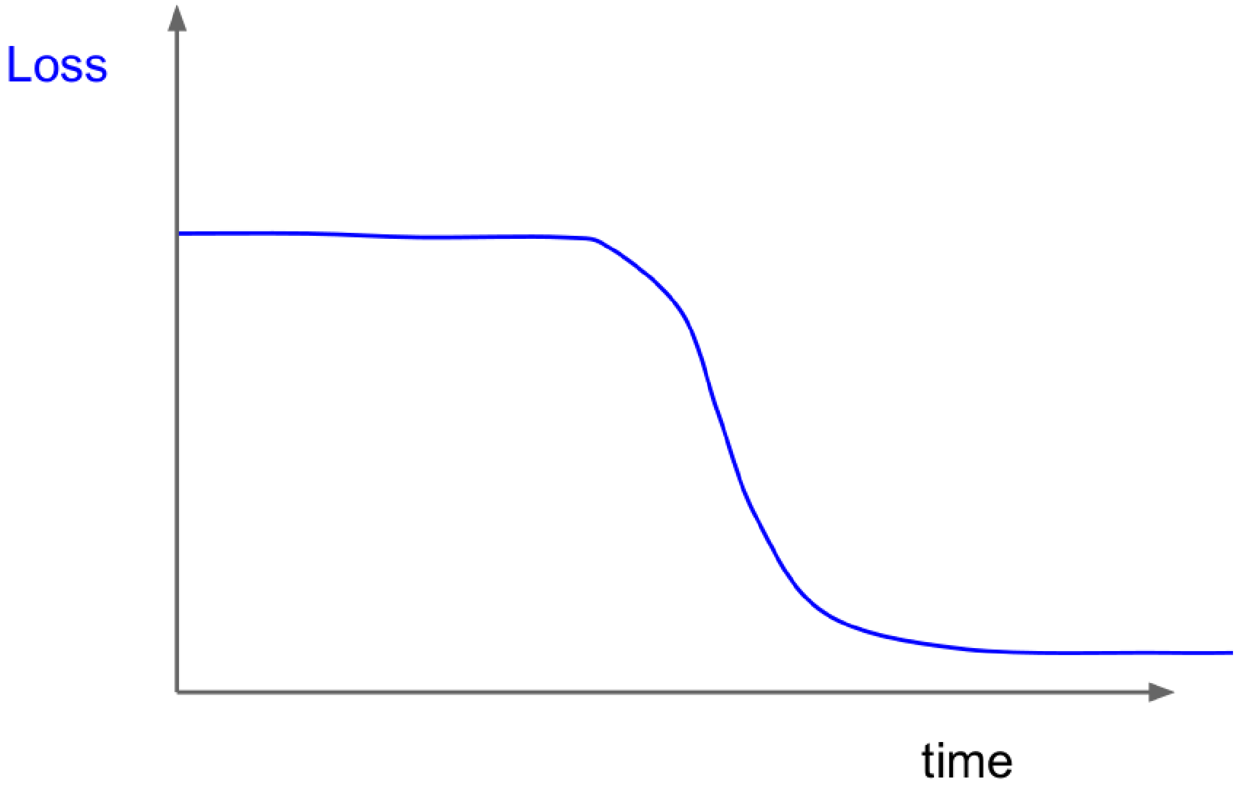
Random Search for Hyper-Parameter Optimization
Bergstra and Bengio, 2012

Some advices from Andrej Karpathy (CS 231n)

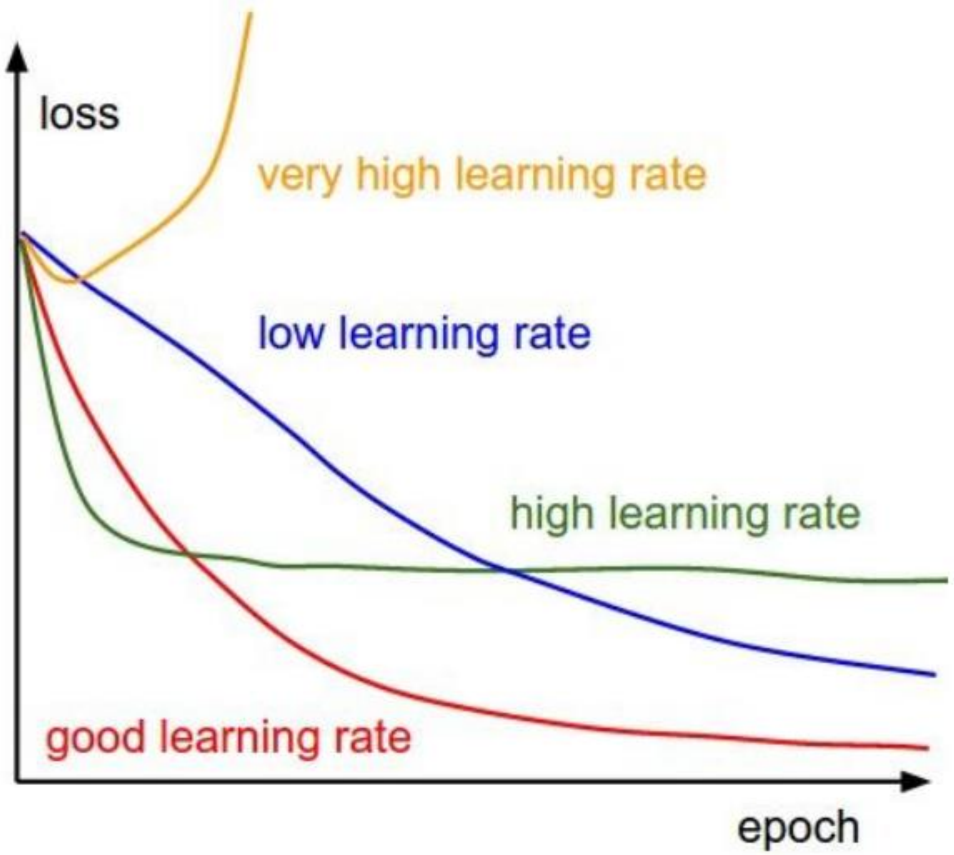
- Coarse to fine tuning
  - run a few epochs for rough search
  - Longer run for finer search
- Break out early if cost > 3 * original cost
- Use log space instead of linear space for LR and reg (weight decay)

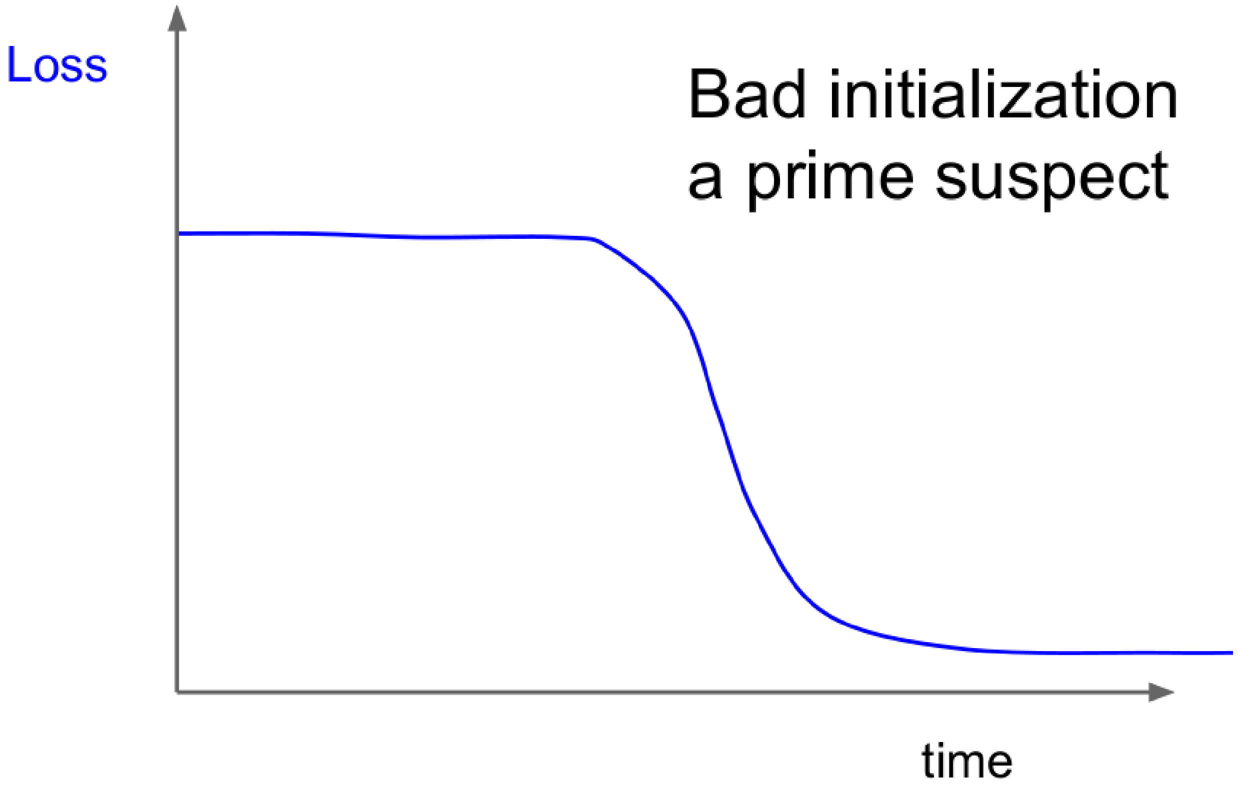
Some advices from Andrej Karpathy (CS 231n)





Some advices from Andrej Karpathy (CS 231n)





Weight and biases (W&B) with Lightning

```python
from pytorch_lightning.loggers import WandbLogger
from pytorch_lightning import Trainer

wandb_logger = WandbLogger()
trainer = Trainer(logger=wandb_logger)
```

```
pip install wandb

wandb login
```

Automatically log hyper parameter

```
config={'n_hidden':5}
wandb.init(project='barebone',name='fast',config=config)

lightmodule = MyLightningModule(wandb.config)
wandb_logger = WandbLogger()
trainer = pl.Trainer(accelerator='gpu', devices=1, max_epochs=10,
        default_root_dir="./lightning-example", logger=wandb_logger)

trainer.fit(model=lightmodule, train_dataloaders=my_dataloader)
```

```
class MyLightningModule(pl.LightningModule):
    def __init__(self, config):
        super().__init__()
        self.net = Net(n_feature=1, n_hidden=config.n_hidden, n_output=1)
```

Hyperparameter sweep

```
def train_model():
    wandb.init(project="sweep")
    config=wandb.config
    wandb_logger = WandbLogger()
    data = MyDataModule(config)
    module = MyLightningModule(config)

    wandb_logger.watch(module.net)

    trainer = pl.Trainer(accelerator='gpu', devices=1, max_epochs=10,
        default_root_dir="./lightning-example", logger=wandb_logger)
    trainer.fit(module, data)
```

Hyperparameter sweep

```
class MyDataModule(pl.LightningDataModule):
    def __init__(self, config):
        super().__init__()
        N=10000
        x = torch.unsqueeze(torch.linspace(-1, 1, N), dim=1)
        y = x.pow(2) + config.noise*torch.rand(x.size())
        self.my_dataset = MyDataset(x,y)
        print(config.noise)

    def train_dataloader(self):
        return DataLoader(self.my_dataset,batch_size=100,shuffle=True)

    def val_dataloader(self):
        return DataLoader(self.my_dataset,batch_size=100,shuffle=False)
```

Hyperparameter sweep

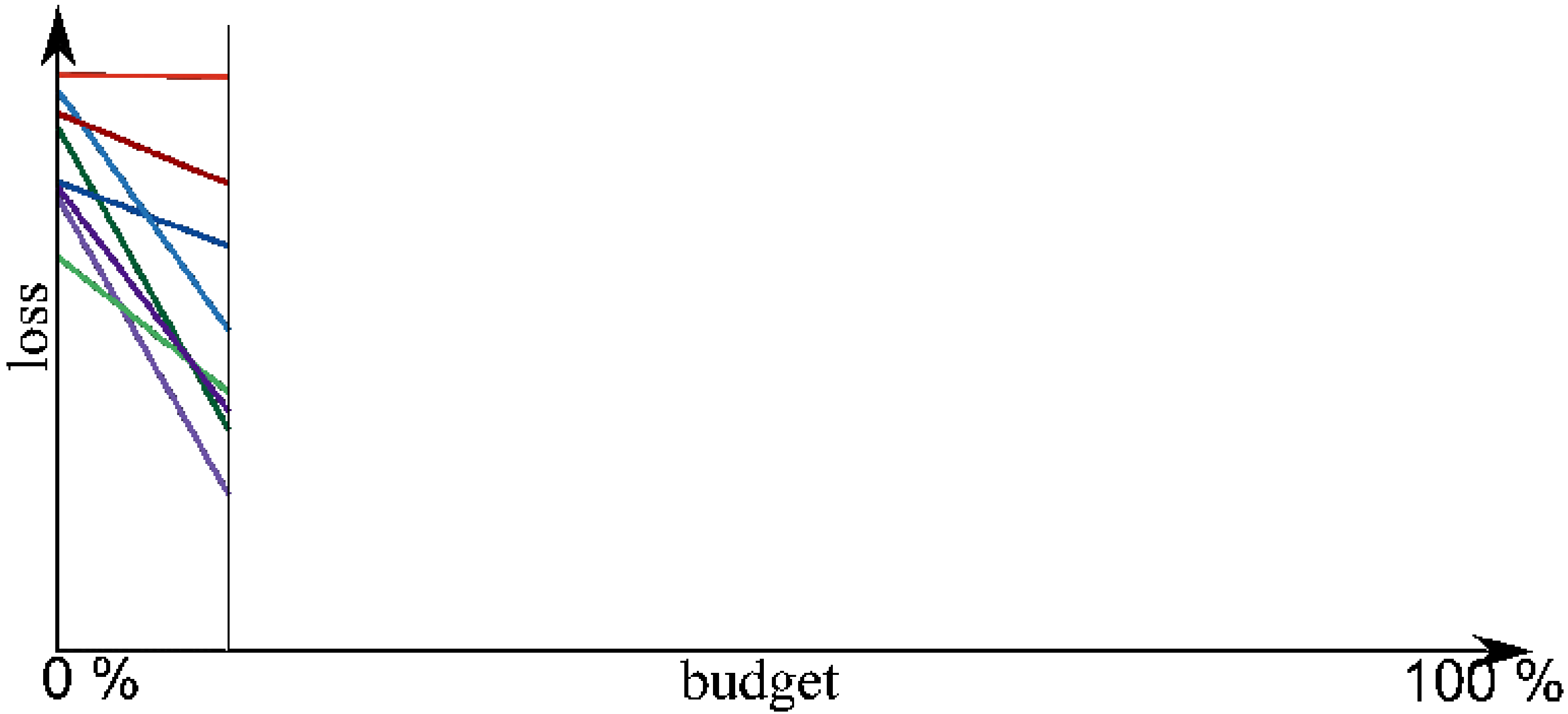
```
if __name__ == '__main__':
    sweep_config = {
        'method': 'random',
        'name': 'first_sweep',
        'metric': {
            'goal': 'minimize',
            'name': 'validation_loss'
        },
        'parameters': {
            'n_hidden': {'values': [2,3,5,10]},
            'lr': {'max': 1.0, 'min': 0.0001},
            'noise': {'max': 1.0, 'min': 0.}
        }
    }

    sweep_id=wandb.sweep(sweep_config, project="test_sweep")
    wandb.agent(sweep_id=sweep_id, function=train_model, count=5)
```

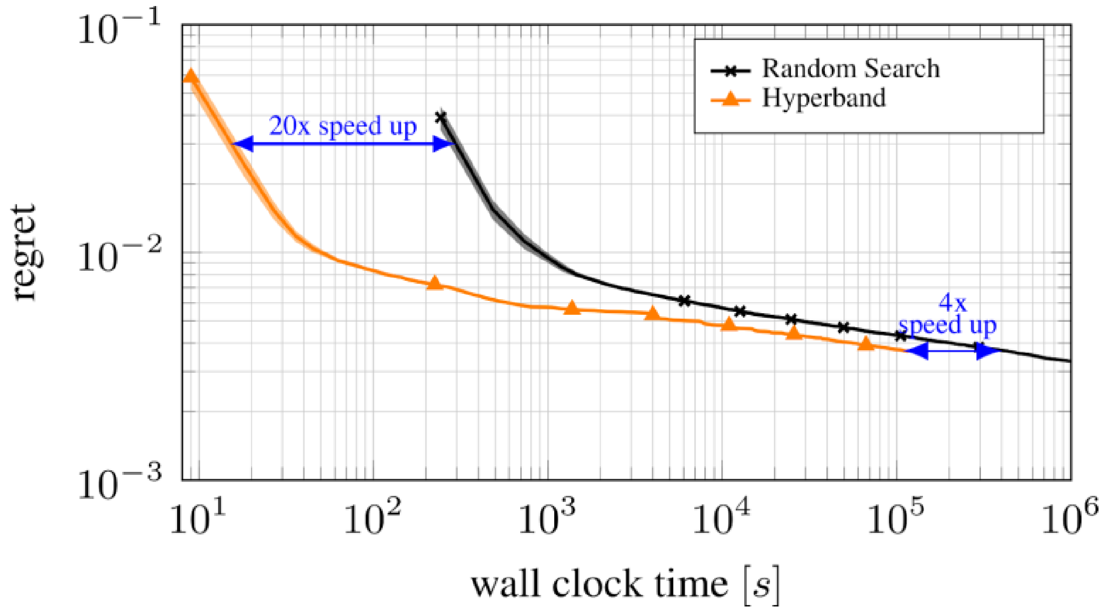
Warning!!!

- W&B sweep conflict with PyTorch Lightning's save_hyperparameters method
- Do NOT use save_hyperparameters() with W&B
  - Took me a week to figure it out. Didn't mention in the documentation
  - The use of save_hyperparameters() in lightning is quite confusing to me, maybe you guys can dig more. But it seems that if you are using W&B, probably there is no need of save_hyperparameters()

Successive halving

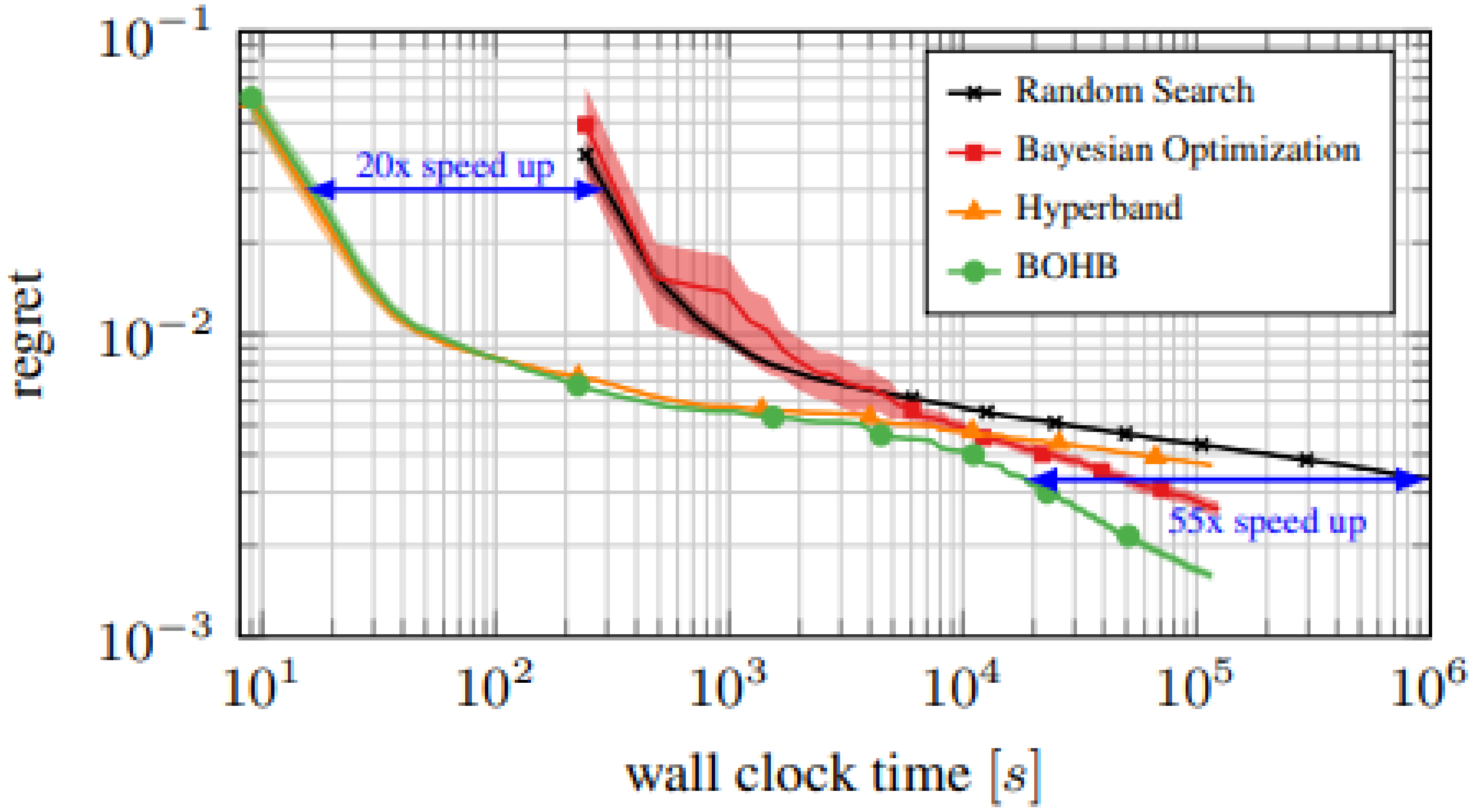


Hyperband



- Given a fixed budget B, it is not clear how many initial configurations n should be used for successive halving
- Consider several possible values of n for a fixed B
  - in essence performing a grid search over feasible value of n

BOHB: Bayesian optimization and Hyperband



Use Bayesian optimization in later stage

AUTOML

The group (from the University of Freiburg) that invented BOHB along group from the University of Hannover have created several open-source tools for AutoML

- Several of the packages are for hyperparameter tuning
  - Such as HpBandSter, which is used by Ray Tune
- The latest version is known as SMAC3

Hyperparameter tuning with SMAC3

```
from ConfigSpace import ConfigurationSpace
from ConfigSpace.hyperparameters import UniformFloatHyperparameter,CategoricalHyperparameter

configspace = ConfigurationSpace()

n_hidden=CategoricalHyperparameter("n_hidden", [1,2,3,5,10])
lr=UniformFloatHyperparameter("lr", 1e-5, 1, log=True)
noise = UniformFloatHyperparameter("noise", 0, 5)
configspace.add_hyperparameters([noise,lr,n_hidden])

# Provide meta data for the optimization
scenario = Scenario({
    "run_obj": "quality",  # Optimize quality (alternatively runtime)
    "runcount-limit": 10,  # Max number of function evaluations (the more the better)
    "cs": configspace
})

smac = SMAC4BB(scenario=scenario, tae_runner=train_model)
best_found_config = smac.optimize()
```

Warning!!!

If you use Ubuntu (20.04 or 22.04) and virtualenv, don't use --system-site-packages

ConfigSpace appears to conflict with --system-site-packages

Jupyter-notebook tips and traps

- Useful hotkeys
  - Esc a/b: insert cells before/after
  - Esc m: change cell to markup
  - Esc y: change cell to code
  - Esc shift-m: merge with below
  - Esc ctrl-shift-'-': split from here
- Jupyter-notebook is very convenient but ...
  - Beware of unintended global variables
    - Esc-00 is your friend
  - When you are really stuck debugging, tidy things up and copy only necessary code to new notebook
    - Things usually will clear up

Summary

- Try out PyTorch lightning (especially if you just start from scratch)
  - Easier to maintain along the way (if you didn't break anything)
  - Some learning curve if you need detailed control (need callbacks)
- Try out W&B or other similar loggers
  - W&B is free for academic use
  - Don't mix pl.save_hyperparameters() with W&B
  - Hyperparameter sweeping is convenient (even tho not the state of the art)
    - Don't use grid search
- Try out AutoML SMAC3
  - Not sure if it will work with loggers like W&B
  - Don't use --system-site-packages if you use virtualenv in Ubuntu (20.04, 22.04)
  - In theory, state-of-the-art hyperparameter tuning (didn't test enough personally)
- Recommend to start with PyTorch->lightning->W&B, each additional layer makes it harder to debug. You may need to work on all different levels of abstraction