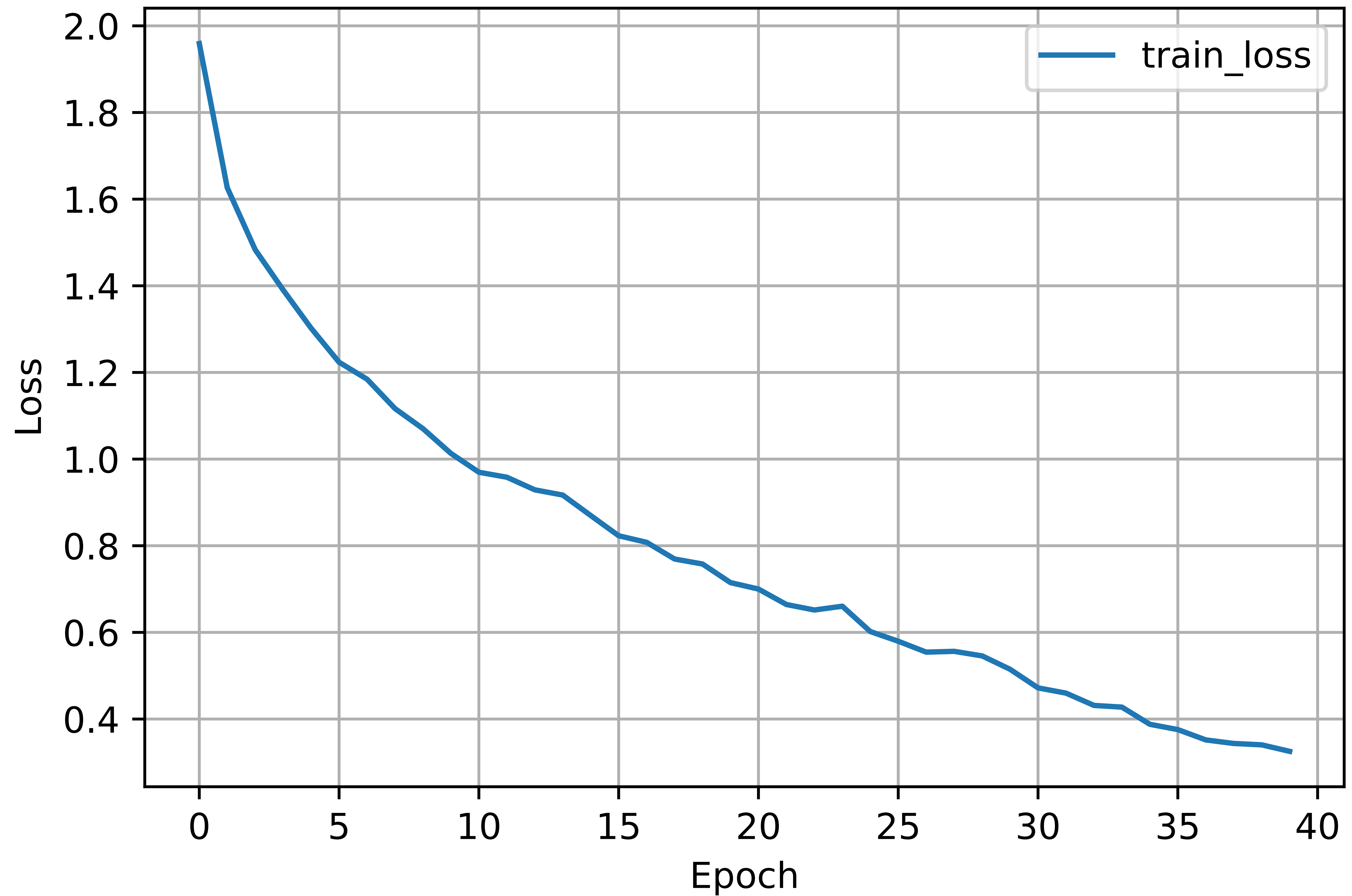
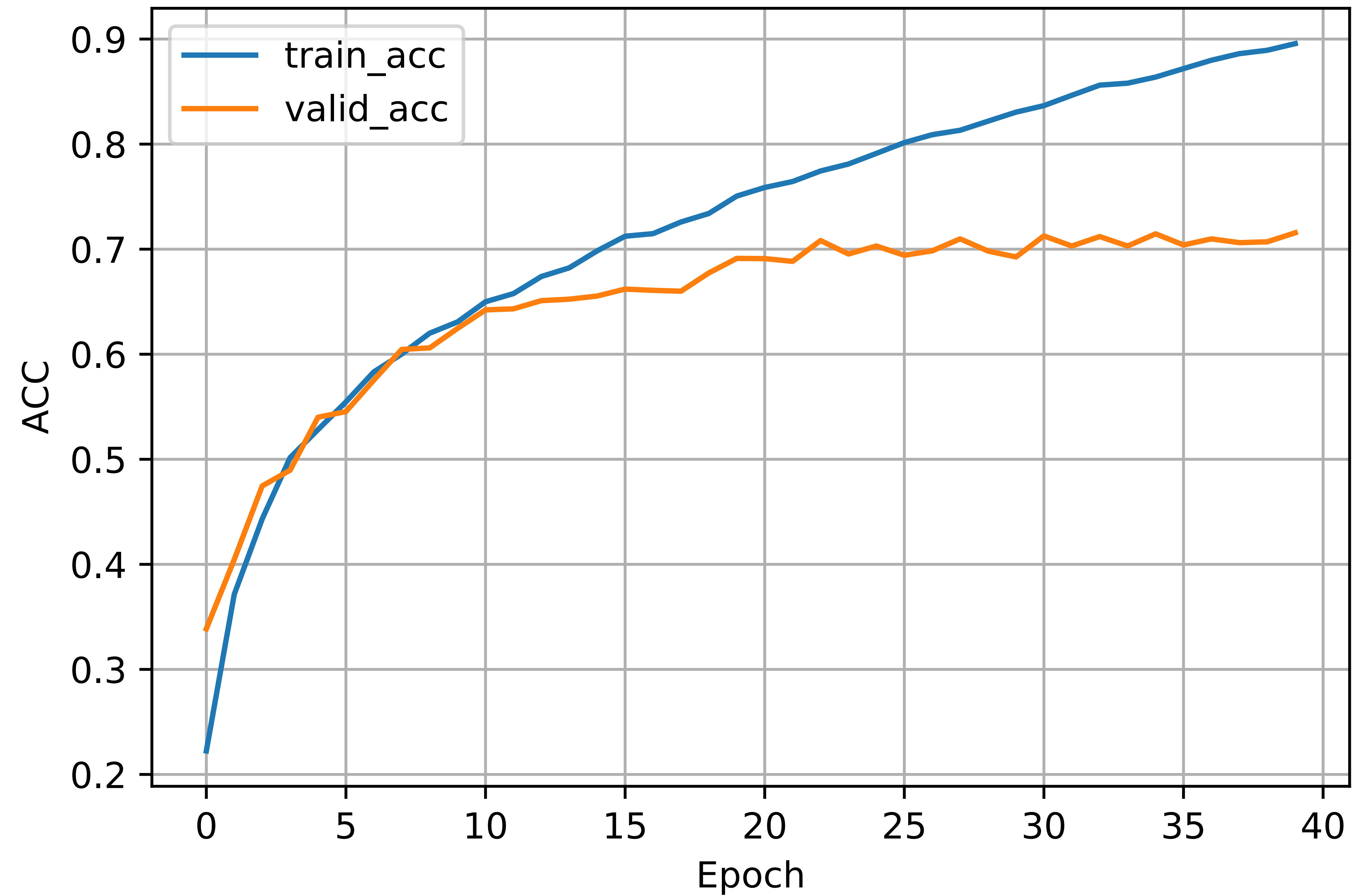
Bonus: Advanced features & techniques

Logging

Does the training loss go down?



Is the model overfitting?



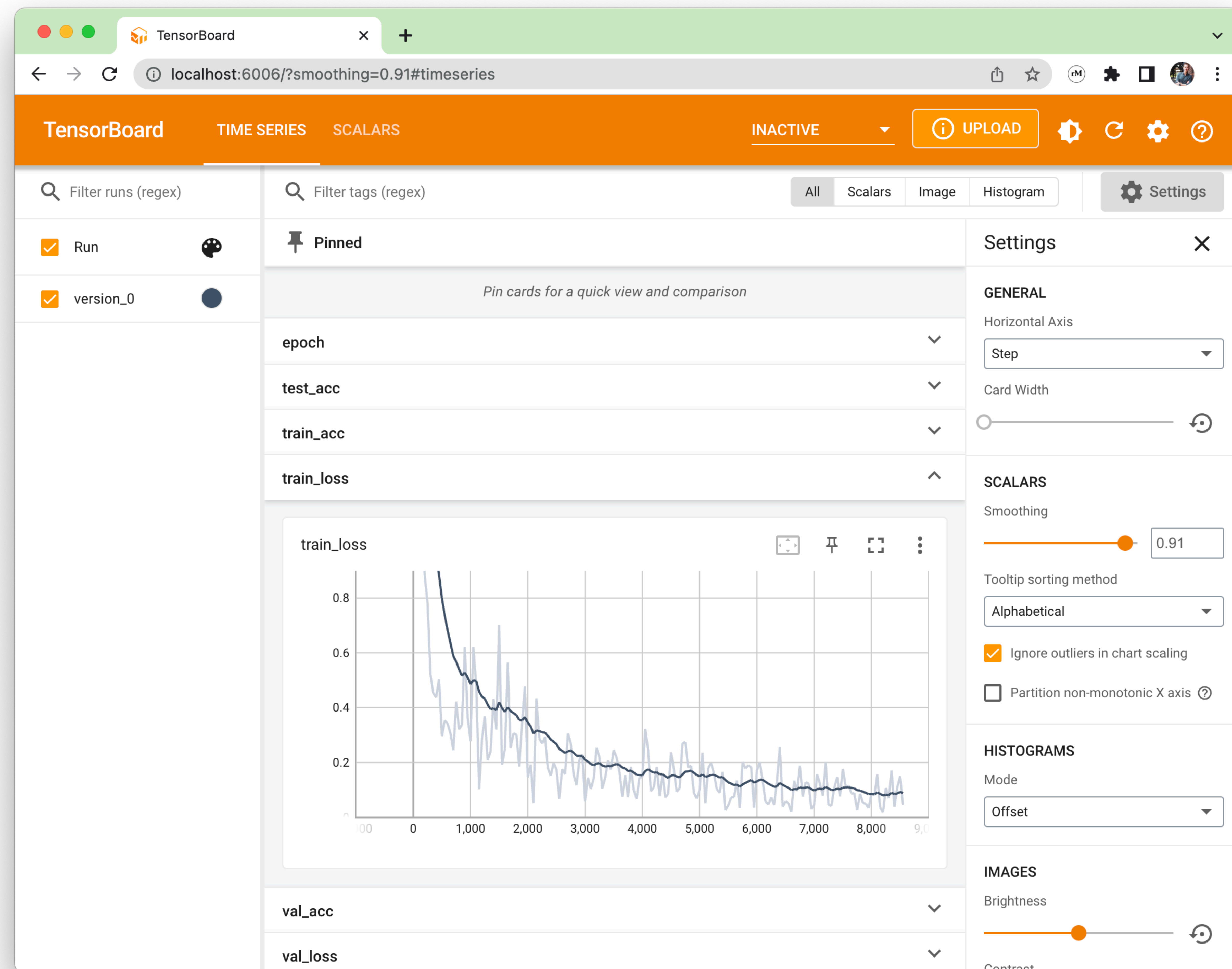
Saving hyperparameters

```
class LightningModel(L.LightningModule):
    def __init__(self, model, learning_rate):
        super().__init__()

        self.learning_rate = learning_rate
        self.model = model

        # Save settings and hyperparameters to the log directory
        # but skip the model parameters
        self.save_hyperparameters(ignore=["model"])

        self.train_acc = torchmetrics.Accuracy()
        self.val_acc = torchmetrics.Accuracy()
        self.test_acc = torchmetrics.Accuracy()

    def forward(self, x):
        return self.model(x)

    ...
```

Currently, the following loggers are supported:

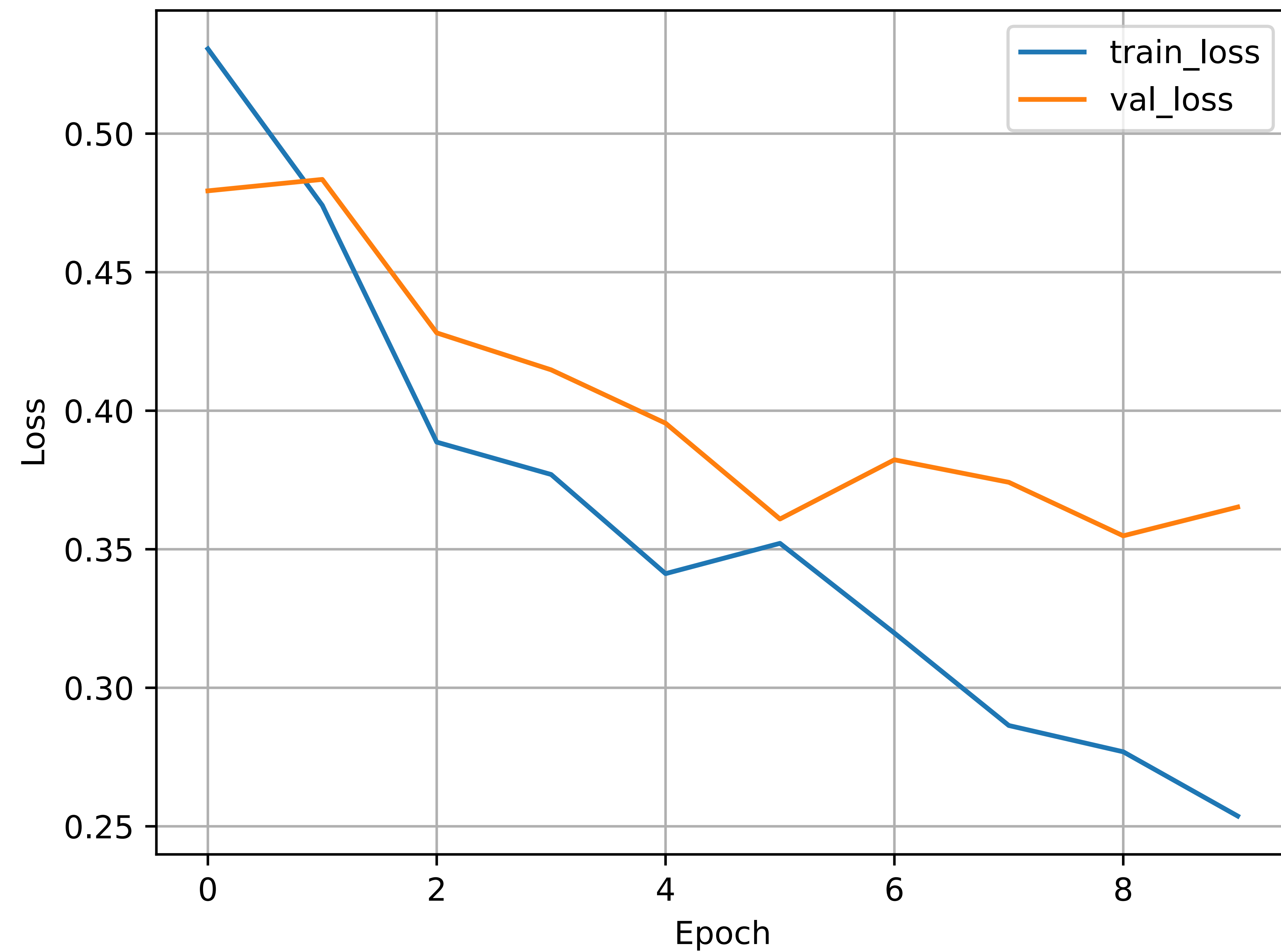
`CometLogger`	Track your parameters, metrics, source code and more using Comet.
`CSVLogger`	Log to local file system in yaml and CSV format.
`MLFlowLogger`	Log using MLflow.
`NeptuneLogger`	Log using Neptune.
`TensorBoardLogger`	Log to local file system in TensorBoard format. ← default setting
`WandbLogger`	Log using Weights and Biases.

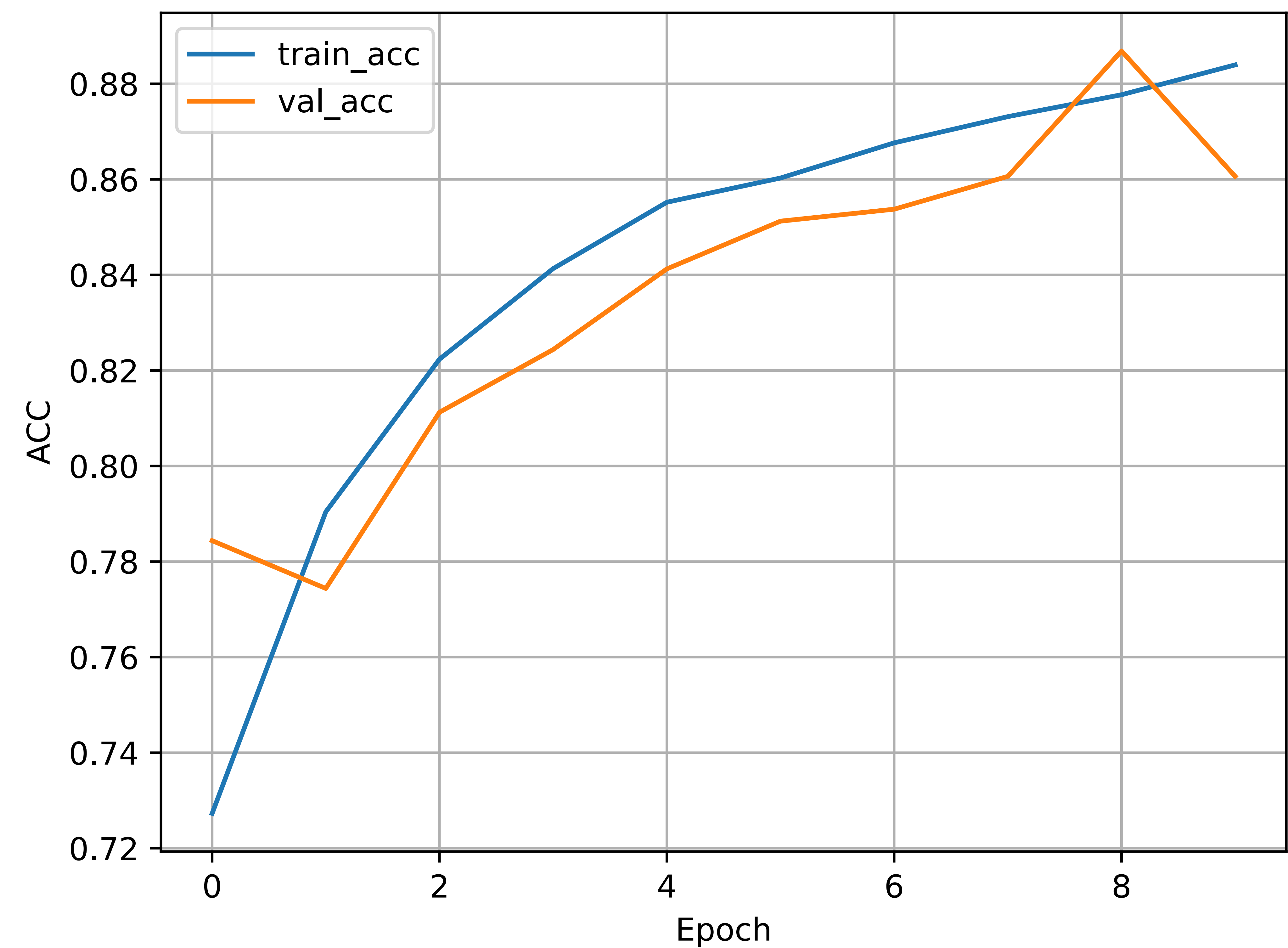
```
Desktop — tensorboard --logdir=~/Desktop/lightning_logs/ — tensorboard — python3.9 ~/miniforg...
(base) ➜  Desktop tensorboard --logdir=~/Desktop/lightning_logs/
TensorFlow installation not found - running with reduced feature set.
Serving TensorBoard on localhost; to expose to the network, use a proxy or pass
--bind_all
TensorBoard 2.10.0 at http://localhost:6006/ (Press CTRL+C to quit)
```



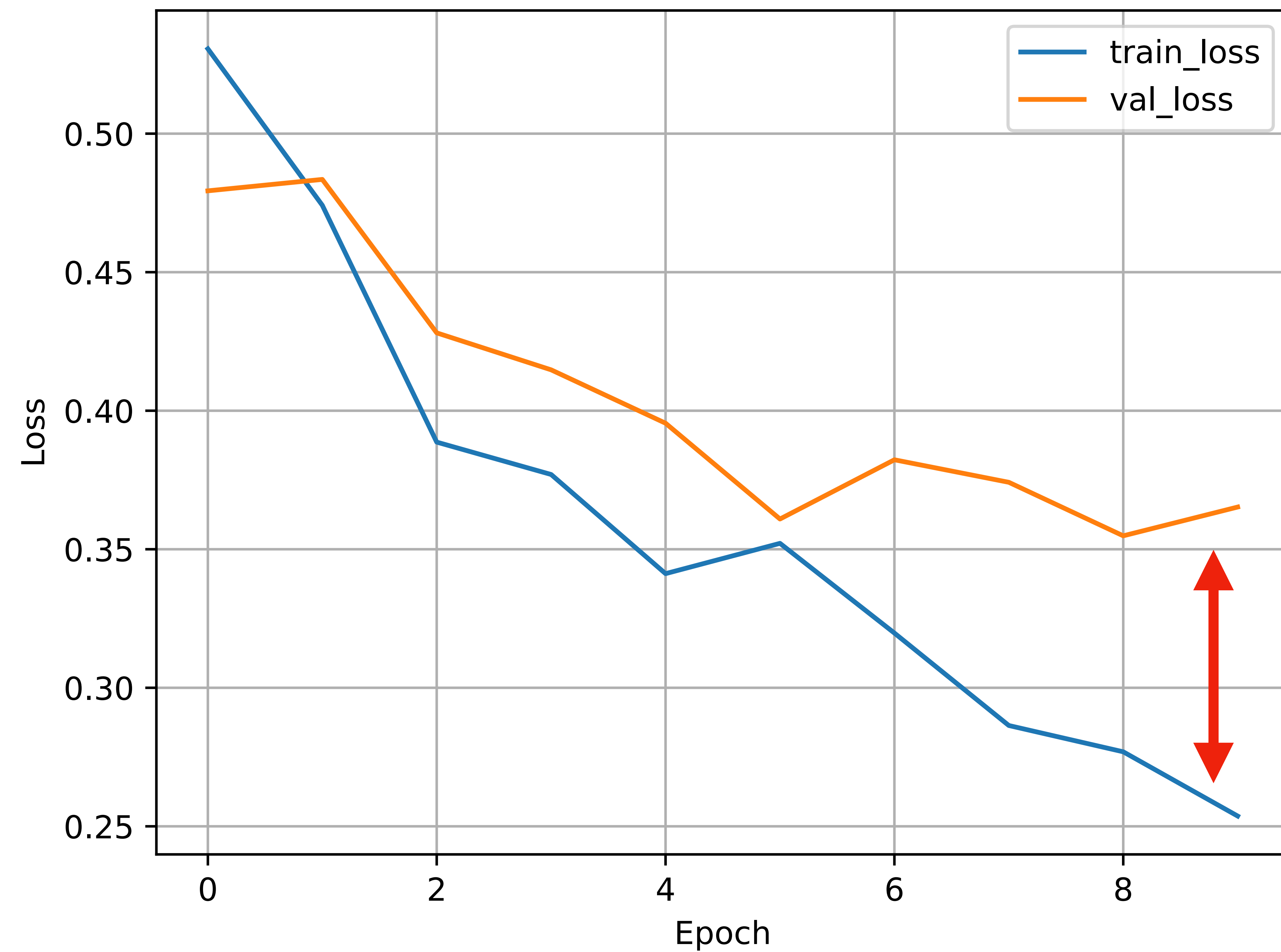
Checkpointing

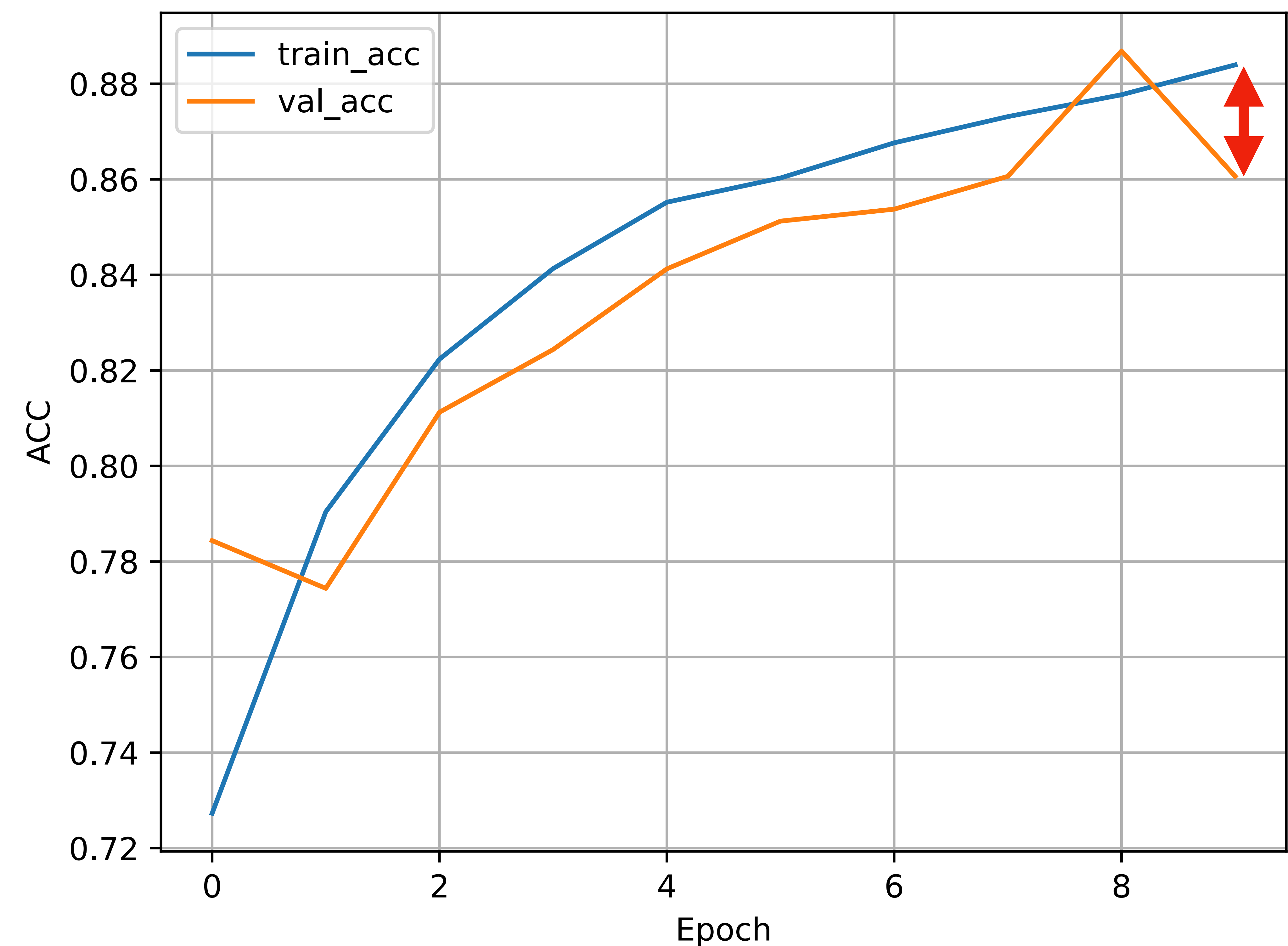
Suppose we trained a model ...





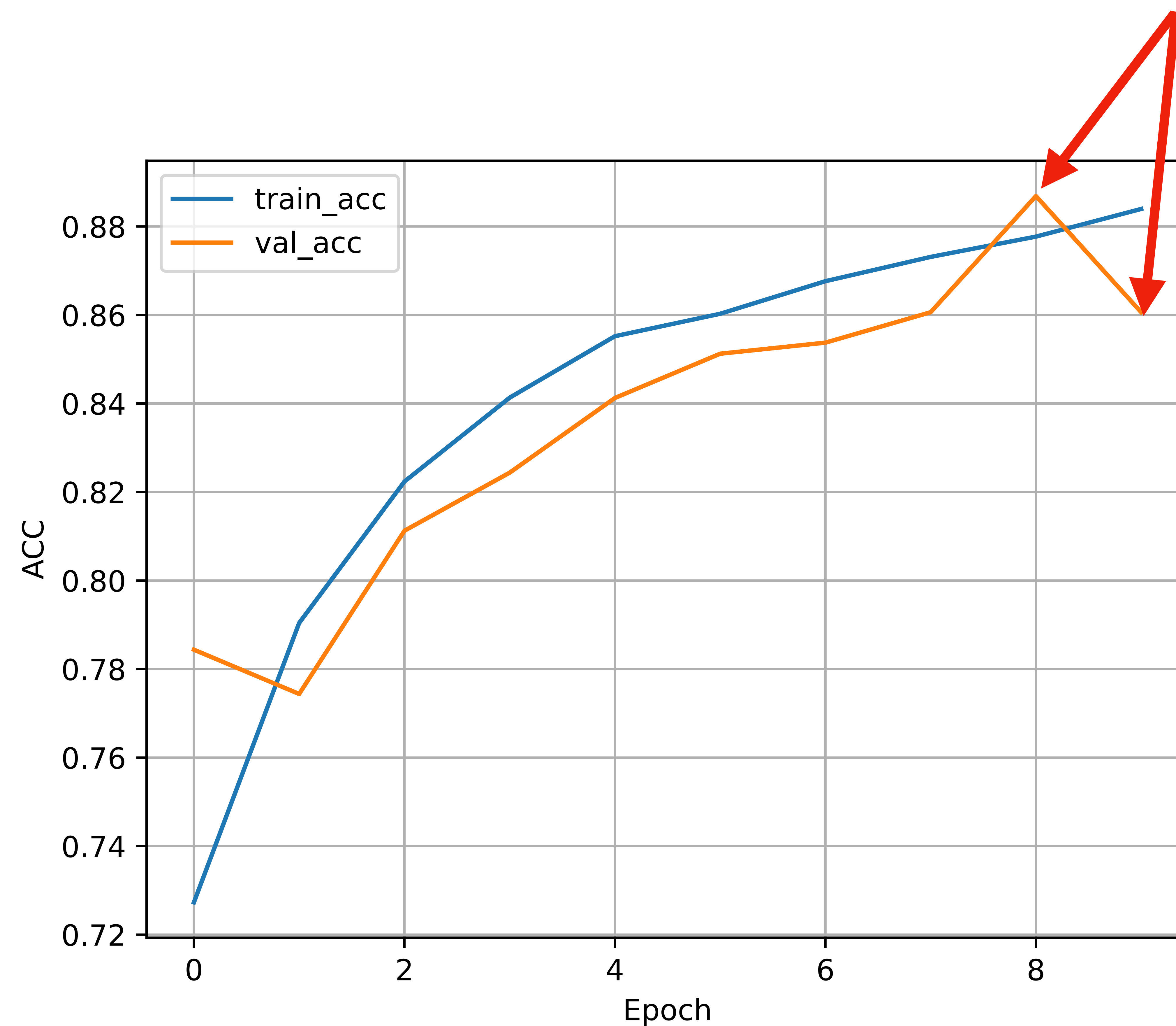
It looks like it becomes worse due to overfitting





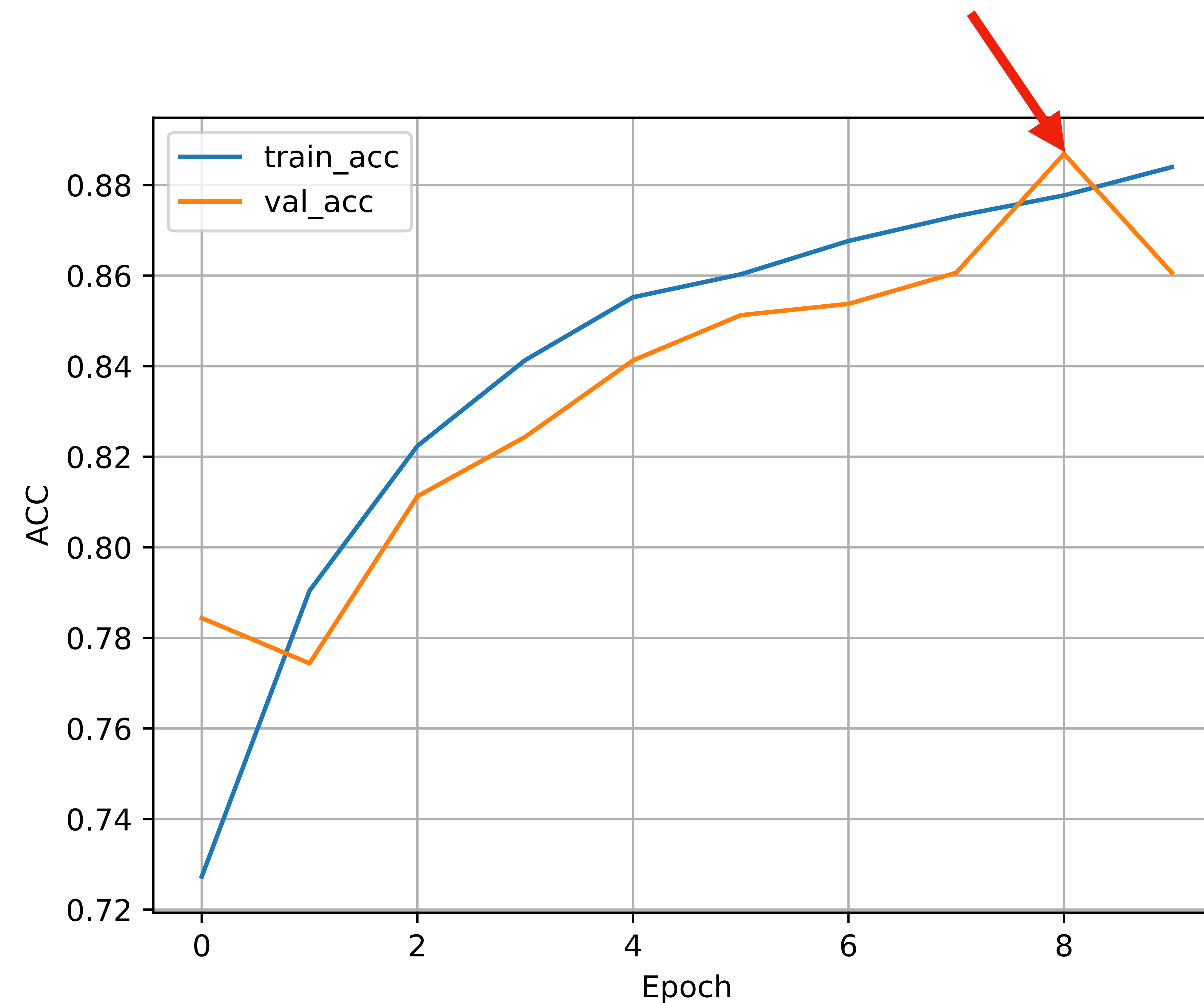
We will talk about strategies to mitigate overfitting later ...

Now, the model from the last epoch is likely not the best



Should we rerun the training with 1 less epoch? (~ early stopping)

We can save the "best" model during training



Define a ModelCheckpoint callback

```
from lightning.pytorch.callbacks import ModelCheckpoint

callbacks = [
    ModelCheckpoint(save_top_k=1, mode="max", monitor="val_acc", save_last=True)
]
```

Define a ModelCheckpoint callback

```
from lightning.pytorch.callbacks import ModelCheckpoint

callbacks = [
    ModelCheckpoint(save_top_k=1, mode="max", monitor="val_acc", save_last=True)
]
```

Define a ModelCheckpoint callback

```
from lightning.pytorch.callbacks import ModelCheckpoint

callbacks = [
    ModelCheckpoint(save_top_k=1, mode="max", monitor="val_acc", save_last=True)
]
```

Save the best model based on maximizing the validation accuracy

Why does this work?

```
from lightning.pytorch.callbacks import ModelCheckpoint

callbacks = [
    ModelCheckpoint(save_top_k=1, mode="max", monitor="val_acc", save_last=True)
]
```

```
class LightningModel(L.LightningModule):
    def __init__(self, model, learning_rate):
        super().__init__()

        self.learning_rate = learning_rate
        self.model = model

    # ...

    def validation_step(self, batch, batch_idx):
        loss, true_labels, predicted_labels = self._shared_step(batch)

        self.log("val_loss", loss, prog_bar=True)
        self.val_acc(predicted_labels, true_labels)
        self.log("val_acc", self.val_acc, prog_bar=True)
```

We track the "val_acc" during training

Access the "best" model after training

```
trainer.test(model=lightning_model, datamodule=dm, ckpt_path="best")
```

Test metric	DataLoader 0
test_acc	0.8622499704360962

```
[{'test_acc': 0.8622499704360962}]
```

```python
trainer.test(model=lightning_model, datamodule=dm, ckpt_path="best")
```

Test metric	DataLoader 0
test_acc	0.8622499704360962

```
[{'test_acc': 0.8622499704360962}]
```

```python
trainer.test(model=lightning_model, datamodule=dm, ckpt_path="last")
```

Test metric	DataLoader 0
test_acc	0.8557500243186951

```
[{'test_acc': 0.8557500243186951}]
```

Making Code Reproducible

Generating a Random Number in Python

```
[1]: import random

[2]: random.random()

[2]: 0.5216761588732212
```

(This will look different on your computer)

```
[1]: import random

[2]: random.random()

[2]: 0.5216761588732212

[3]: random.random()

[3]: 0.1550530569802705
```

(This will look different on your computer)

Let's seed the random number generator

Setting a Random Number Seed

```
[4]: random.seed(1)
     random.random()

[4]: 0.13436424411240122
```

(This will look the same on your computer)

Now, Let's Call Random() Repeatedly

```
[4]: random.seed(1)
     random.random()

[4]: 0.13436424411240122

[5]: random.random()

[5]: 0.8474337369372327
```

(This will look the same on your computer)

The execution sequence becomes reproducible when we seed the random number generator

```
[4]: random.seed(1)
     random.random()

[4]: 0.13436424411240122

[5]: random.random()

[5]: 0.8474337369372327

[6]: random.seed(1)
     random.random()

[6]: 0.13436424411240122

[7]: random.random()

[7]: 0.8474337369372327
```

The Same Concept Applies To PyTorch

Some common sources of randomness are

1. Model weight initialization
2. Dataset shuffling and augmentation
3. Nondeterministic algorithms
4. (GPU) Hardware and drivers

Most deep learning training scripts are not reproducible across different environments

Bu trained models themselves (usually) behave deterministically

Controlling common sources of randomness

1. Model weight initialization
2. Dataset shuffling and augmentation
3. Nondeterministic algorithms
4. (GPU) Hardware and drivers

Exercise

Make code deterministic