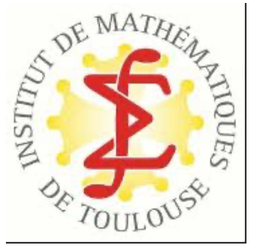
Make Your Training Loop Private: A Hands-On Introduction to Differential Privacy

Clément Lalanne - ANITI Days 2026



ANITI

```
def train_nonprivate(model, epochs=epochs, bs=batch_size, lr=lr):
    opt = optim.SGD(model.parameters(), lr=lr)
    losses = []
    for _ in range(epochs):
        idx = torch.randperm(n)
        for i in range(0, n, bs):
            b = idx[i:i+bs]
            xb, yb = X[b], y[b]
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
            losses.append(loss.item())
    return losses
```

```python
def train_nonprivate(model, epochs=epochs, bs=batch_size, lr=lr):
    opt = optim.SGD(model.parameters(), lr=lr)
    losses = []
    for _ in range(epochs):
        idx = torch.randperm(n)
        for i in range(0, n, bs):
            b = idx[i:i+bs]
            xb, yb = X[b], y[b]
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
            losses.append(loss.item())
    return losses
```

Can I deploy this model without leaking too much user data?

```python
def train_nonprivate(model, epochs=epochs, bs=batch_size, lr=lr):
    opt = optim.SGD(model.parameters(), lr=lr)
    losses = []
    for _ in range(epochs):
        idx = torch.randperm(n)
        for i in range(0, n, bs):
            b = idx[i:i+bs]
            xb, yb = X[b], y[b]
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
            losses.append(loss.item())
    return losses
```

Can I deploy this model without leaking too much user data?

How can someone recover user data from the model?

- Directly from the weights (e.g., memorization of the dataset in the weights of overparameterized models).
- Simply through black-box access to the model (e.g., “Hey GPT, can you tell me surprising anecdotes that happened to your users?”).

```
def train_nonprivate(model, epochs=epochs, bs=batch_size, lr=lr):
    opt = optim.SGD(model.parameters(), lr=lr)
    losses = []
    for _ in range(epochs):
        idx = torch.randperm(n)
        for i in range(0, n, bs):
            b = idx[i:i+bs]
            xb, yb = X[b], y[b]
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
            losses.append(loss.item())
    return losses
```

Where does the model learn about your data?

```
def train_nonprivate(model, epochs=epochs, bs=batch_size, lr=lr):
    opt = optim.SGD(model.parameters(), lr=lr)
    losses = []
    for _ in range(epochs):
        idx = torch.randperm(n)
        for i in range(0, n, bs):
            b = idx[i:i+bs]
            xb, yb = X[b], y[b]
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
            losses.append(loss.item())
    return losses
```

Where does the model learn about your data?

```python
def train_nonprivate(model, epochs=epochs, bs=batch_size, lr=lr):
    opt = optim.SGD(model.parameters(), lr=lr)
    losses = []
    for _ in range(epochs):
        idx = torch.randperm(n)
        for i in range(0, n, bs):
            b = idx[i:i+bs]
            xb, yb = X[b], y[b]
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
            losses.append(loss.item())
    return losses
```

$$\theta \leftarrow \theta - lr \times \left\{ \frac{1}{batch\_size} \sum_{j \in batch} \nabla l(y_j, model(x_j)) \right\}$$

```python
def train_nonprivate(model, epochs=epochs, bs=batch_size, lr=lr):
    opt = optim.SGD(model.parameters(), lr=lr)
    losses = []
    for _ in range(epochs):
        idx = torch.randperm(n)
        for i in range(0, n, bs):
            b = idx[i:i+bs]
            xb, yb = X[b], y[b]
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
            losses.append(loss.item())
    return losses
```

$$\theta \leftarrow \theta - lr \times \left\{ \frac{1}{batch\_size} \sum_{j \in batch} \nabla l(y_j, model(x_j)) \right\}$$

$$\sum_{j \in batch \setminus \{your\_data\}} \nabla l(y_j, model(x_j)) + \mathbf{1}_{your\_data \in batch} \nabla l(y_{your\_data}, model(x_{your\_data}))$$

```python
def train_nonprivate(model, epochs=epochs, bs=batch_size, lr=lr):
    opt = optim.SGD(model.parameters(), lr=lr)
    losses = []
    for _ in range(epochs):
        idx = torch.randperm(n)
        for i in range(0, n, bs):
            b = idx[i:i+bs]
            xb, yb = X[b], y[b]
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
            losses.append(loss.item())
    return losses
```

$$\theta \leftarrow \theta - lr \times \left\{ \frac{1}{batch\_size} \sum_{j \in batch} \nabla l(y_j, model(x_j)) \right\}$$

$$\sum_{j \in batch \setminus \{your\_data\}} \nabla l(y_j, model(x_j)) + \mathbf{1}_{your\_data \in batch} \nabla l(y_{your\_data}, model(x_{your\_data}))$$

To understand what the model learns from you, ask a *counterfactual*: **would the trained model look essentially the same if your data were someone else's?**

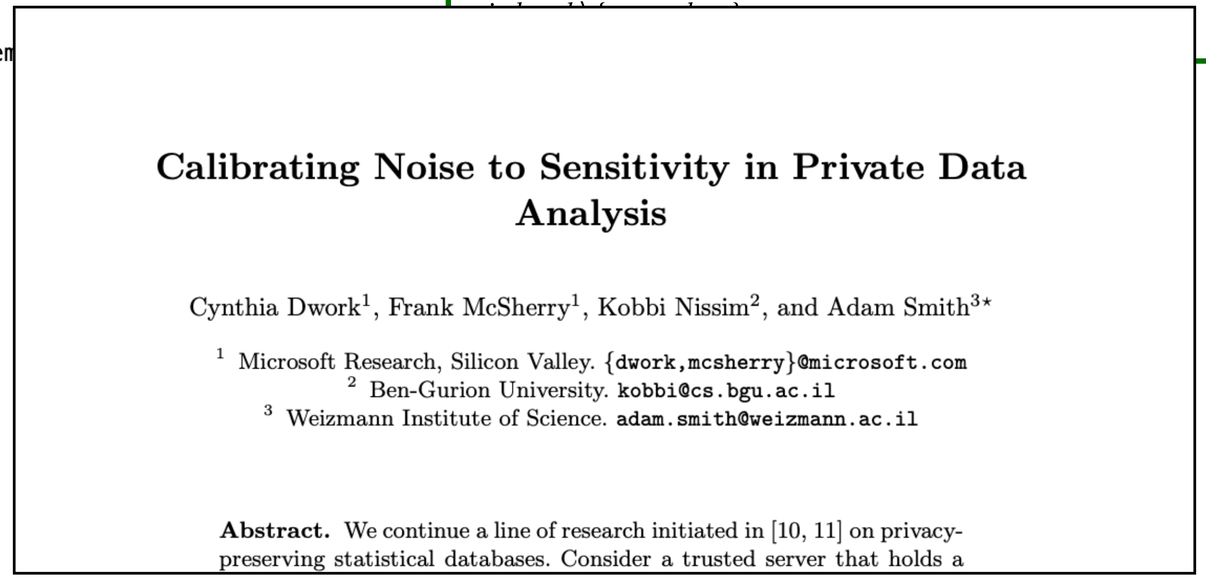
Differential privacy makes this precise by requiring training to be **stable to the inclusion/removal of any single user**.

As a result, the model captures patterns shared across many users and *filters out information that is unique to one person*.

```python
def train_nonprivate(model, epochs=epochs, bs=batch_size, lr=lr):
    opt = optim.SGD(model.parameters(), lr=lr)
    losses = []
    for _ in range(epochs):
        idx = torch.randperm(n)
        for i in range(0, n, bs):
            b = idx[i:i+bs]
            xb, yb = X[b], y[b]
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
            losses.append(loss.item
    return losses
```

$$\theta \leftarrow \theta - lr \times \left\{ \frac{1}{batch\_size} \sum_{j \in batch} \nabla l(y_j, model(x_j)) \right\}$$

$$\sum \quad \nabla l(y_j, model(x_j)) + \mathbf{1}_{your\_data \in batch} \nabla l(y_{your\_data}, model(x_{your\_data}))$$



Calibrating Noise to Sensitivity in Private Data Analysis

Cynthia Dwork[1], Frank McSherry[1], Kobbi Nissim[2], and Adam Smith[3⋆]

[1] Microsoft Research, Silicon Valley. {dwork,mcsherry}@microsoft.com
[2] Ben-Gurion University. kobbi@cs.bgu.ac.il
[3] Weizmann Institute of Science. adam.smith@weizmann.ac.il

Abstract. We continue a line of research initiated in [10, 11] on privacy-preserving statistical databases. Consider a trusted server that holds a

```python
def train_nonprivate(model, epochs=epochs, bs=batch_size, lr=lr):
    opt = optim.SGD(model.parameters(), lr=lr)
    losses = []
    for _ in range(epochs):
        idx = torch.randperm(n)
        for i in range(0, n, bs):
            b = idx[i:i+bs]
            xb, yb = X[b], y[b]
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
            losses.append(loss.it
    return losses
```

$$\theta \leftarrow \theta - lr \times \left\{ \frac{1}{batch\_size} \sum_{j \in batch} \nabla l(y_j, model(x_j)) \right\}$$

$$\sum_{j \in batch \setminus \{your\_data\}} \nabla l(y_j, model(x_j)) + \mathbf{1}_{your\_data \in batch} \nabla l(y_{your\_data}, model(x_{your\_data}))$$

2017 Gödel Prize:

The 2017 Gödel Prize is awarded to **Cynthia Dwork, Frank McSherry, Kobbi Nissim and Adam Smith** for their work on Differential Privacy in the paper:

Calibrating Noise to Sensitivity in Private Data Analysis,
Journal of Privacy and Confidentiality, Volume 7, Issue 3, 2016
(preliminary version in *Theory of Cryptography, TCC 2006*).

Cynthia Dwork, Frank McSherry, Kobbi Nissim and Adam Smith will receive the 2017 Gödel Prize at the 49th Annual ACM Symposium on Theory of Computing, to be held from 19-23 June, 2017 in Montreal, Canada.

Differential privacy is a powerful theoretical model for dealing with the privacy of statistical data. By

Calibrating Noise to

Cynthia Dwork[1], Frank McSh

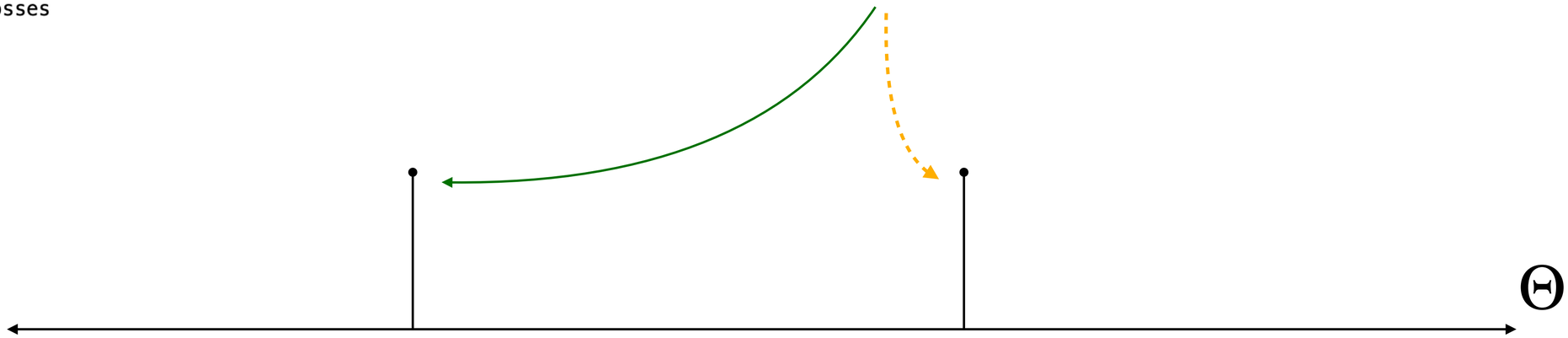
[1] Microsoft Research, Silicon
[2] Ben-Gurion University. kobbi@cs.bgu.ac.il
[3] Weizmann Institute of Science. adam.smith@weizmann.ac.il

Abstract. We continue a line of research initiated in [10, 11] on privacy-preserving statistical databases. Consider a trusted server that holds a

```python
def train_nonprivate(model, epochs=epochs, bs=batch_size, lr=lr):
    opt = optim.SGD(model.parameters(), lr=lr)
    losses = []
    for _ in range(epochs):
        idx = torch.randperm(n)
        for i in range(0, n, bs):
            b = idx[i:i+bs]
            xb, yb = X[b], y[b]
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
            losses.append(loss.item())
    return losses
```

$$\theta \leftarrow \theta - lr \times \left\{ \frac{1}{batch\_size} \sum_{j \in batch} \nabla l(y_j, model(x_j)) \right\}$$

$$\sum_{j \in batch \setminus \{your\_data\}} \nabla l(y_j, model(x_j)) + \mathbf{1}_{your\_data \in batch} \nabla l(y_{your\_data}, model(x_{your\_data}))$$



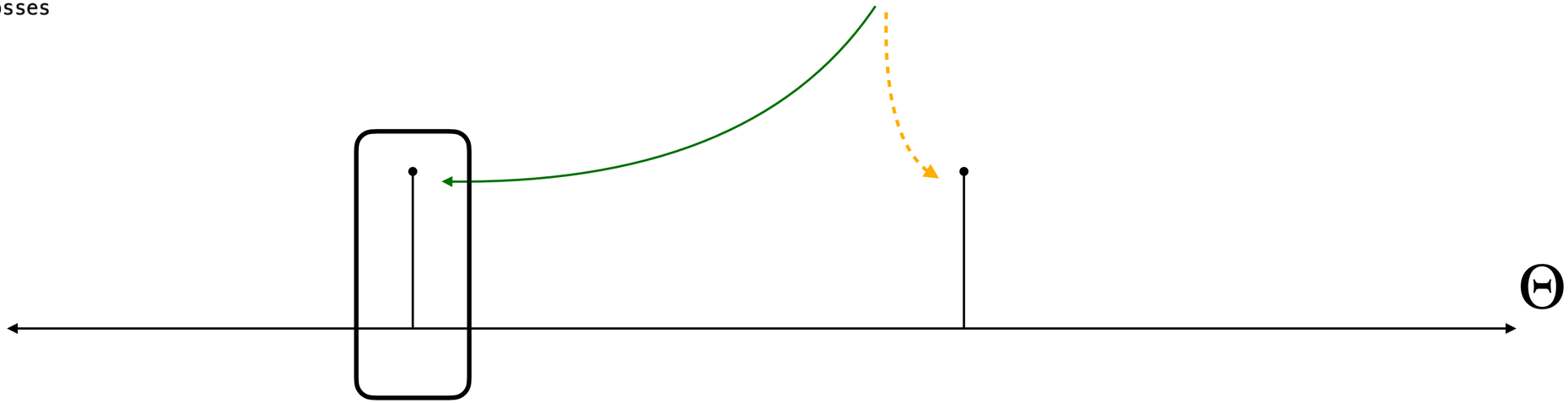
Update direction with your data

Update direction if your data were replaced by Chuck Norris's data

```python
def train_nonprivate(model, epochs=epochs, bs=batch_size, lr=lr):
    opt = optim.SGD(model.parameters(), lr=lr)
    losses = []
    for _ in range(epochs):
        idx = torch.randperm(n)
        for i in range(0, n, bs):
            b = idx[i:i+bs]
            xb, yb = X[b], y[b]
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
            losses.append(loss.item())
    return losses
```

$$\theta \leftarrow \theta - lr \times \left\{ \frac{1}{batch\_size} \sum_{j \in batch} \nabla l(y_j, model(x_j)) \right\}$$

$$\sum_{j \in batch \setminus \{your\_data\}} \nabla l(y_j, model(x_j)) + \mathbf{1}_{your\_data \in batch} \nabla l(y_{your\_data}, model(x_{your\_data}))$$

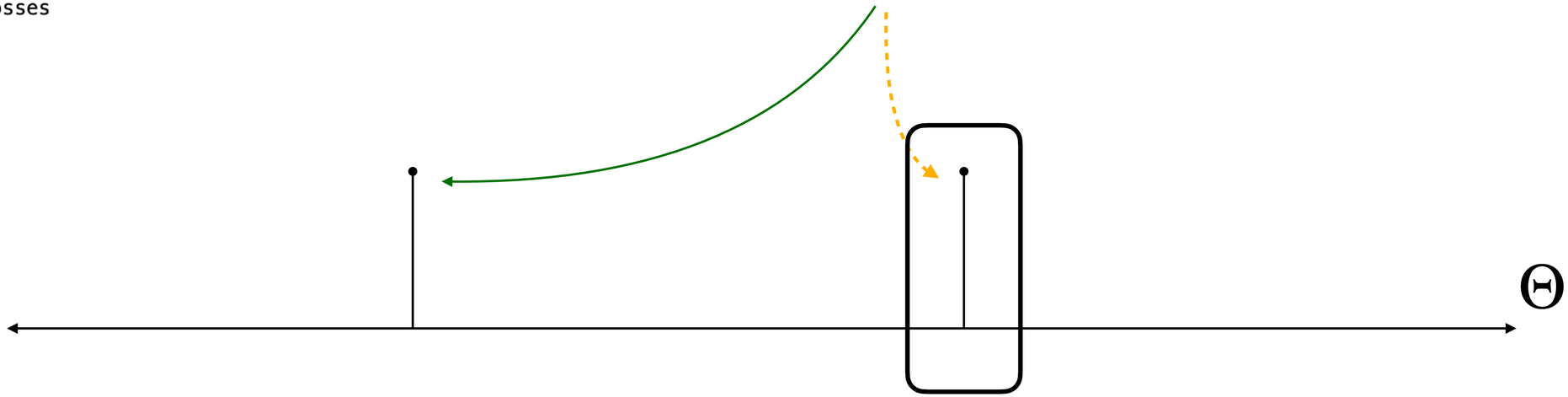


If this is the update direction, I know for a fact that your data was part of the training set.

```python
def train_nonprivate(model, epochs=epochs, bs=batch_size, lr=lr):
    opt = optim.SGD(model.parameters(), lr=lr)
    losses = []
    for _ in range(epochs):
        idx = torch.randperm(n)
        for i in range(0, n, bs):
            b = idx[i:i+bs]
            xb, yb = X[b], y[b]
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
            losses.append(loss.item())
    return losses
```

$$\theta \leftarrow \theta - lr \times \left\{ \frac{1}{batch\_size} \sum_{j \in batch} \nabla l(y_j, model(x_j)) \right\}$$

$$\sum_{j \in batch \setminus \{your\_data\}} \nabla l(y_j, model(x_j)) + \mathbf{1}_{your\_data \in batch} \nabla l(y_{your\_data}, model(x_{your\_data}))$$

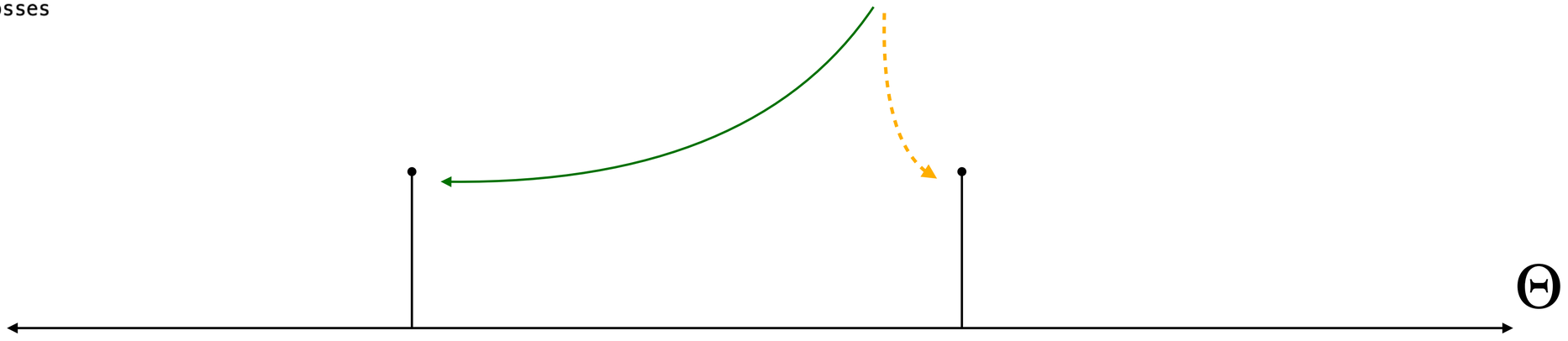


If this is the update direction, I know for a fact that it was Chuck Norris's instead.

```python
def train_nonprivate(model, epochs=epochs, bs=batch_size, lr=lr):
    opt = optim.SGD(model.parameters(), lr=lr)
    losses = []
    for _ in range(epochs):
        idx = torch.randperm(n)
        for i in range(0, n, bs):
            b = idx[i:i+bs]
            xb, yb = X[b], y[b]
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
            losses.append(loss.item())
    return losses
```

$$\theta \leftarrow \theta - lr \times \left\{ \frac{1}{batch\_size} \sum_{j \in batch} \nabla l(y_j, model(x_j)) \right\}$$

$$\sum_{j \in batch \setminus \{your\_data\}} \nabla l(y_j, model(x_j)) + \mathbf{1}_{your\_data \in batch} \nabla l(y_{your\_data}, model(x_{your\_data}))$$

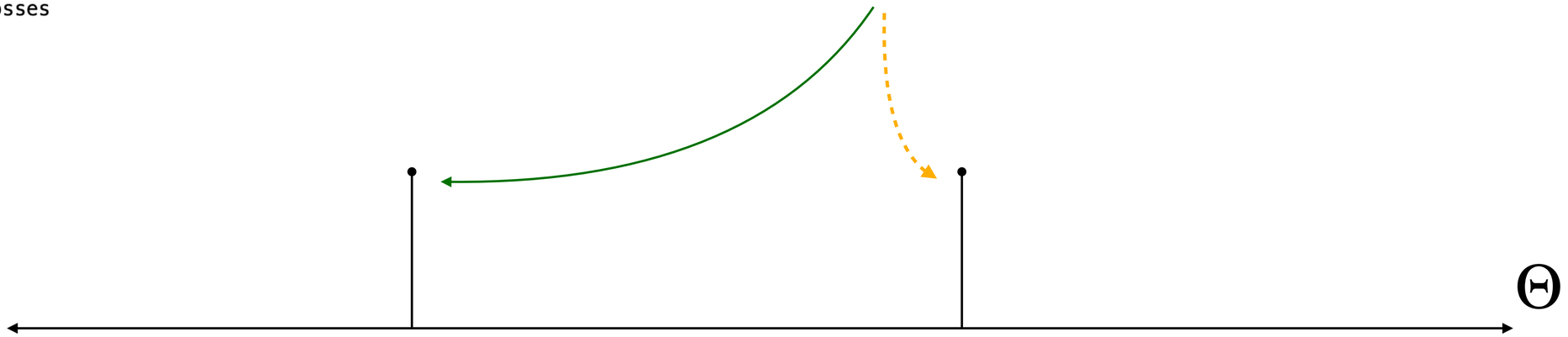


This is problematic: the model has learned **too much user-specific noise**.

```python
def train_nonprivate(model, epochs=epochs, bs=batch_size, lr=lr):
    opt = optim.SGD(model.parameters(), lr=lr)
    losses = []
    for _ in range(epochs):
        idx = torch.randperm(n)
        for i in range(0, n, bs):
            b = idx[i:i+bs]
            xb, yb = X[b], y[b]
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
            losses.append(loss.item())
    return losses
```

$$\theta \leftarrow \theta - lr \times \left\{ \frac{1}{batch\_size} \sum_{j \in batch} \nabla l(y_j, model(x_j)) \right\}$$

$$\sum_{j \in batch \setminus \{your\_data\}} \nabla l(y_j, model(x_j)) + \mathbf{1}_{your\_data \in batch} \nabla l(y_{your\_data}, model(x_{your\_data}))$$



Differential privacy enforces stability by **perturbing the update direction** so that it **cannot depend strongly on any individual user's data**.

```python
def train_nonprivate(model, epochs=epochs, bs=batch_size, lr=lr):
    opt = optim.SGD(model.parameters(), lr=lr)
    losses = []
    for _ in range(epochs):
        idx = torch.randperm(n)
        for i in range(0, n, bs):
            b = idx[i:i+bs]
            xb, yb = X[b], y[b]
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
            losses.append(loss.item())
    return losses
```

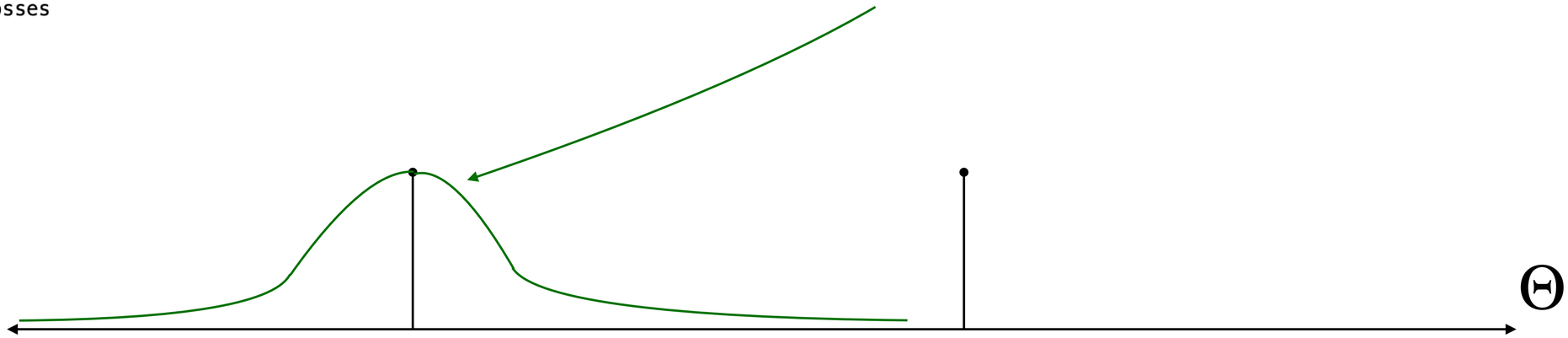
$$\theta \leftarrow \theta - lr \times \left\{ \frac{1}{batch\_size} \left(\sum_{j \in batch} \nabla l(y_j, model(x_j)) + \sigma \mathcal{N}(0,I) \right) \right\}$$

$$\sum_{j \in batch \setminus \{your\_data\}} \nabla l(y_j, model(x_j)) + \mathbf{1}_{your\_data \in batch} \nabla l(y_{your\_data}, model(x_{your\_data})) + \sigma \mathcal{N}(0,I)$$

```python
def train_nonprivate(model, epochs=epochs, bs=batch_size, lr=lr):
    opt = optim.SGD(model.parameters(), lr=lr)
    losses = []
    for _ in range(epochs):
        idx = torch.randperm(n)
        for i in range(0, n, bs):
            b = idx[i:i+bs]
            xb, yb = X[b], y[b]
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
            losses.append(loss.item())
    return losses
```

$$\theta \leftarrow \theta - lr \times \left\{ \frac{1}{batch\_size} \left(\sum_{j \in batch} \nabla l(y_j, model(x_j)) + \sigma \mathcal{N}(0,I) \right) \right\}$$

$$\sum_{j \in batch \setminus \{your\_data\}} \nabla l(y_j, model(x_j)) + \mathbf{1}_{your\_data \in batch} \nabla l(y_{your\_data}, model(x_{your\_data})) + \sigma \mathcal{N}(0,I)$$

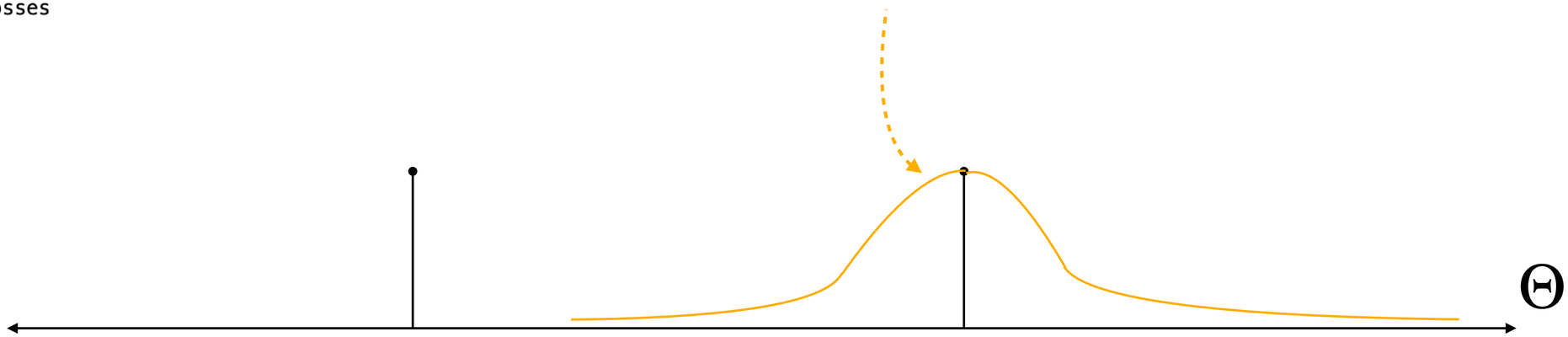


Update direction
with your data

```python
def train_nonprivate(model, epochs=epochs, bs=batch_size, lr=lr):
    opt = optim.SGD(model.parameters(), lr=lr)
    losses = []
    for _ in range(epochs):
        idx = torch.randperm(n)
        for i in range(0, n, bs):
            b = idx[i:i+bs]
            xb, yb = X[b], y[b]
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
            losses.append(loss.item())
    return losses
```

$$\theta \leftarrow \theta - lr \times \left\{ \frac{1}{batch\_size} \left(\sum_{j \in batch} \nabla l(y_j, model(x_j)) + \sigma \mathcal{N}(0,I) \right) \right\}$$

$$\sum_{j \in batch \setminus \{your\_data\}} \nabla l(y_j, model(x_j)) + \mathbf{1}_{your\_data \in batch} \nabla l(y_{your\_data}, model(x_{your\_data})) + \sigma \mathcal{N}(0,I)$$

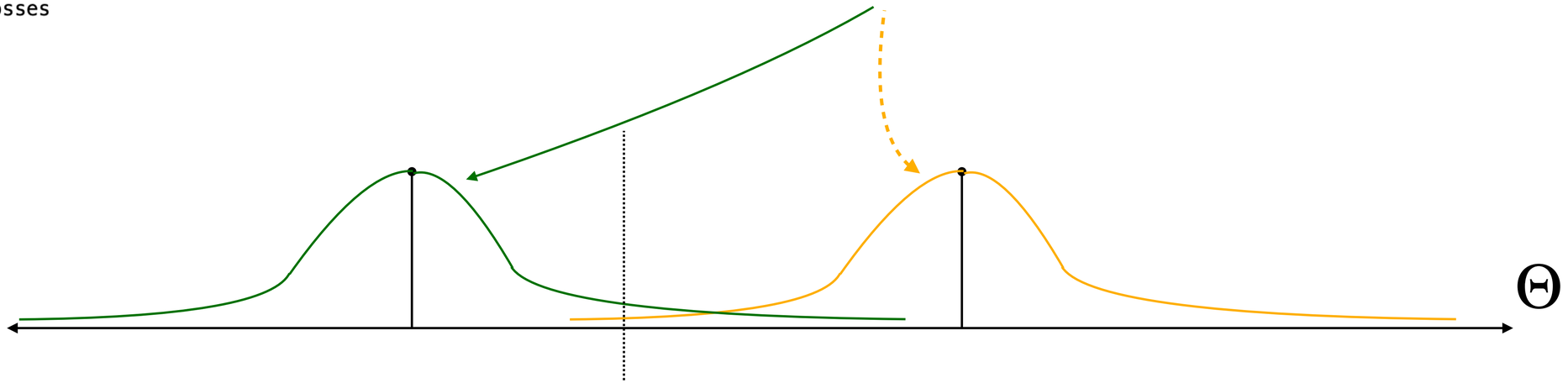


Update direction if your data were replaced by Chuck Norris's data

```python
def train_nonprivate(model, epochs=epochs, bs=batch_size, lr=lr):
    opt = optim.SGD(model.parameters(), lr=lr)
    losses = []
    for _ in range(epochs):
        idx = torch.randperm(n)
        for i in range(0, n, bs):
            b = idx[i:i+bs]
            xb, yb = X[b], y[b]
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
            losses.append(loss.item())
    return losses
```

$$\theta \leftarrow \theta - lr \times \left\{ \frac{1}{batch\_size} \left(\sum_{j \in batch} \nabla l(y_j, model(x_j)) + \sigma \mathcal{N}(0,I) \right) \right\}$$

$$\sum_{j \in batch \setminus \{your\_data\}} \nabla l(y_j, model(x_j)) + \mathbf{1}_{your\_data \in batch} \nabla l(y_{your\_data}, model(x_{your\_data})) + \sigma \mathcal{N}(0,I)$$

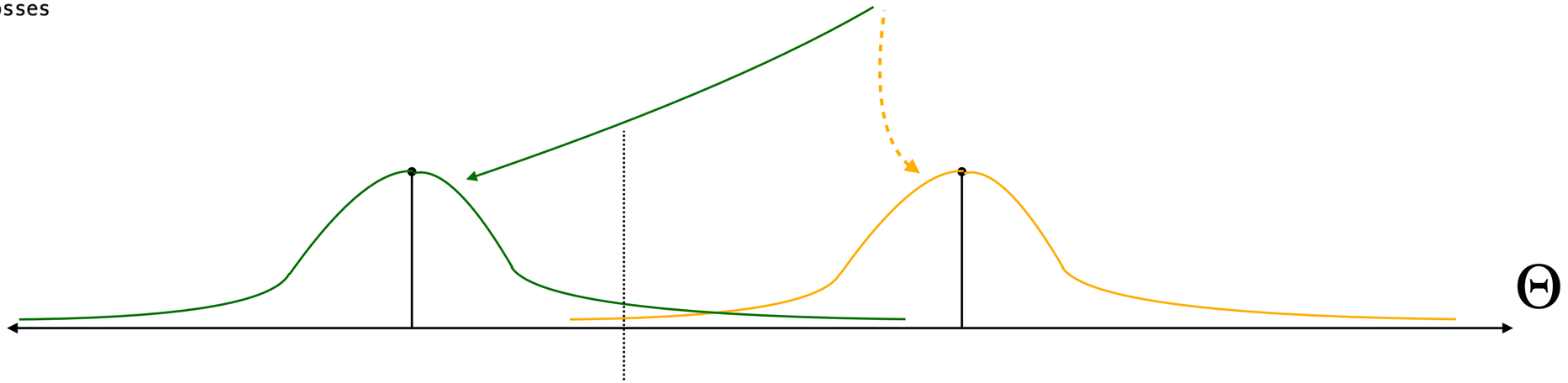


From the observed update direction, it is **impossible to determine exactly** whether your data or Chuck Norris's data was used.

```python
def train_nonprivate(model, epochs=epochs, bs=batch_size, lr=lr):
    opt = optim.SGD(model.parameters(), lr=lr)
    losses = []
    for _ in range(epochs):
        idx = torch.randperm(n)
        for i in range(0, n, bs):
            b = idx[i:i+bs]
            xb, yb = X[b], y[b]
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
            losses.append(loss.item())
    return losses
```

$$\theta \leftarrow \theta - lr \times \left\{ \frac{1}{batch\_size} \left(\sum_{j \in batch} \nabla l(y_j, model(x_j)) + \sigma \mathcal{N}(0,I) \right) \right\}$$

$$\sum_{j \in batch \setminus \{your\_data\}} \nabla l(y_j, model(x_j)) + \mathbf{1}_{your\_data \in batch} \nabla l(y_{your\_data}, model(x_{your\_data})) + \sigma \mathcal{N}(0,I)$$



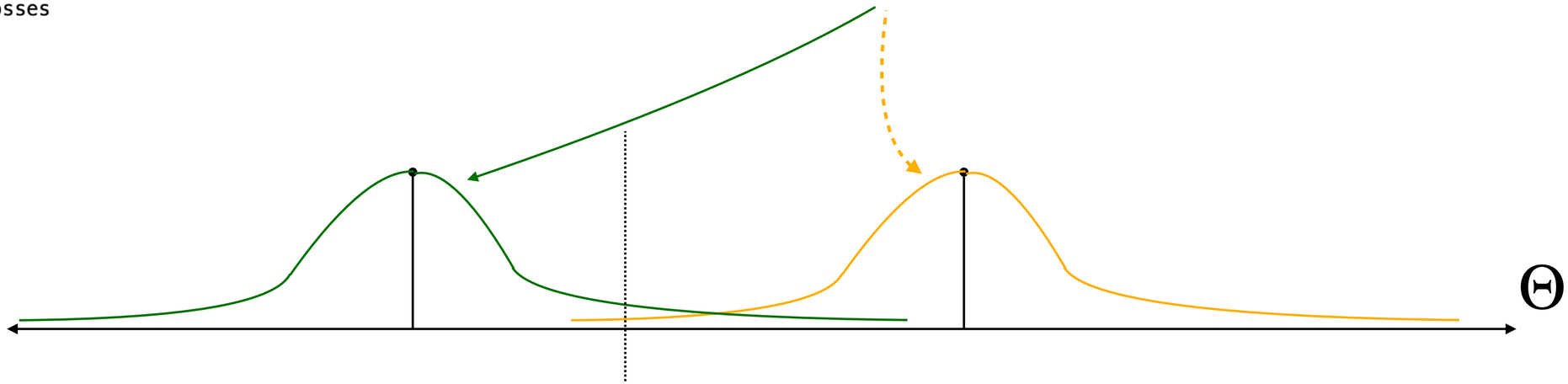
From the observed update direction, it is **impossible to determine exactly** whether your data or Chuck Norris's data was used.

The most I can do is **update a prior belief via the likelihood ratio**, yielding explicit and quantifiable bounds.

```python
def train_nonprivate(model, epochs=epochs, bs=batch_size, lr=lr):
    opt = optim.SGD(model.parameters(), lr=lr)
    losses = []
    for _ in range(epochs):
        idx = torch.randperm(n)
        for i in range(0, n, bs):
            b = idx[i:i+bs]
            xb, yb = X[b], y[b]
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
            losses.append(loss.item())
    return losses
```

$$\theta \leftarrow \theta - lr \times \left\{ \frac{1}{batch\_size} \left(\sum_{j \in batch} \nabla l(y_j, model(x_j)) + \sigma \mathcal{N}(0, I) \right) \right\}$$

$$\sum_{j \in batch \setminus \{your\_data\}} \nabla l(y_j, model(x_j)) + \mathbf{1}_{your\_data \in batch} \nabla l(y_{your\_data}, model(x_{your\_data})) + \sigma \mathcal{N}(0, I)$$

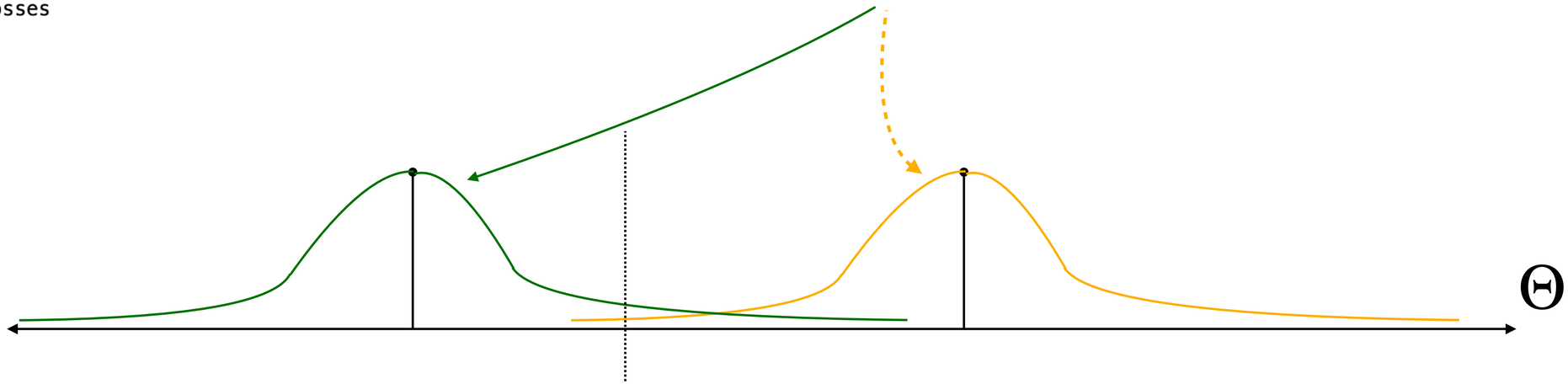


Let's identify the key control knobs and understand how they shape the privacy-utility tradeoff.

```python
def train_nonprivate(model, epochs=epochs, bs=batch_size, lr=lr):
    opt = optim.SGD(model.parameters(), lr=lr)
    losses = []
    for _ in range(epochs):
        idx = torch.randperm(n)
        for i in range(0, n, bs):
            b = idx[i:i+bs]
            xb, yb = X[b], y[b]
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
            losses.append(loss.item())
    return losses
```

$$\theta \leftarrow \theta - lr \times \left\{ \frac{1}{batch\_size} \left(\sum_{j \in batch} \nabla l(y_j, model(x_j)) + \sigma \mathcal{N}(0,I) \right) \right\}$$

$$\sum_{j \in batch \setminus \{your\_data\}} \nabla l(y_j, model(x_j)) + \mathbf{1}_{your\_data \in batch} \nabla l(y_{your\_data}, model(x_{your\_data})) + \sigma \mathcal{N}(0,I)$$

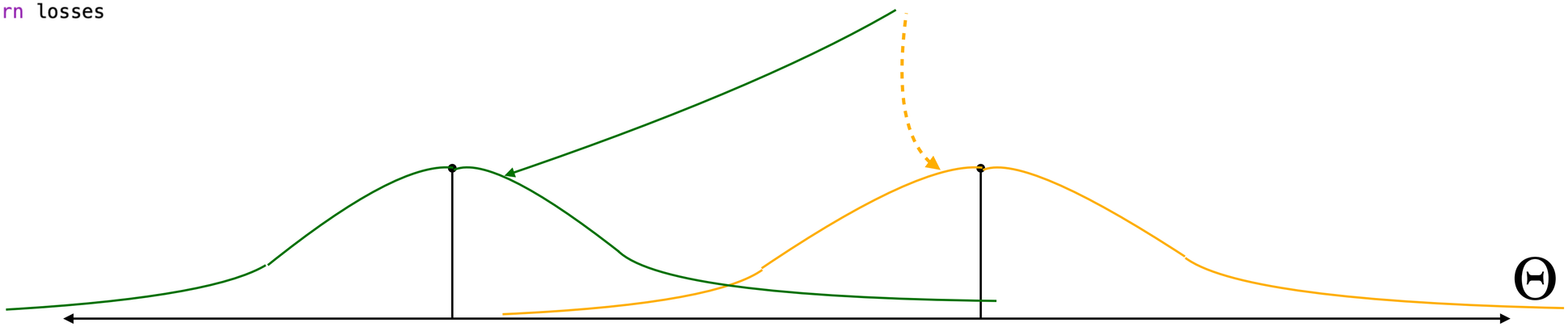


The noise level σ

```python
def train_nonprivate(model, epochs=epochs, bs=batch_size, lr=lr):
    opt = optim.SGD(model.parameters(), lr=lr)
    losses = []
    for _ in range(epochs):
        idx = torch.randperm(n)
        for i in range(0, n, bs):
            b = idx[i:i+bs]
            xb, yb = X[b], y[b]
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
            losses.append(loss.item())
    return losses
```

$$\theta \leftarrow \theta - lr \times \left\{ \frac{1}{batch\_size} \left(\sum_{j \in batch} \nabla l(y_j, model(x_j)) + \sigma \mathcal{N}(0,I) \right) \right\}$$

$$\sum_{j \in batch \setminus \{your\_data\}} \nabla l(y_j, model(x_j)) + \mathbf{1}_{your\_data \in batch} \nabla l(y_{your\_data}, model(x_{your\_data})) + \sigma \mathcal{N}(0,I)$$



The noise level σ

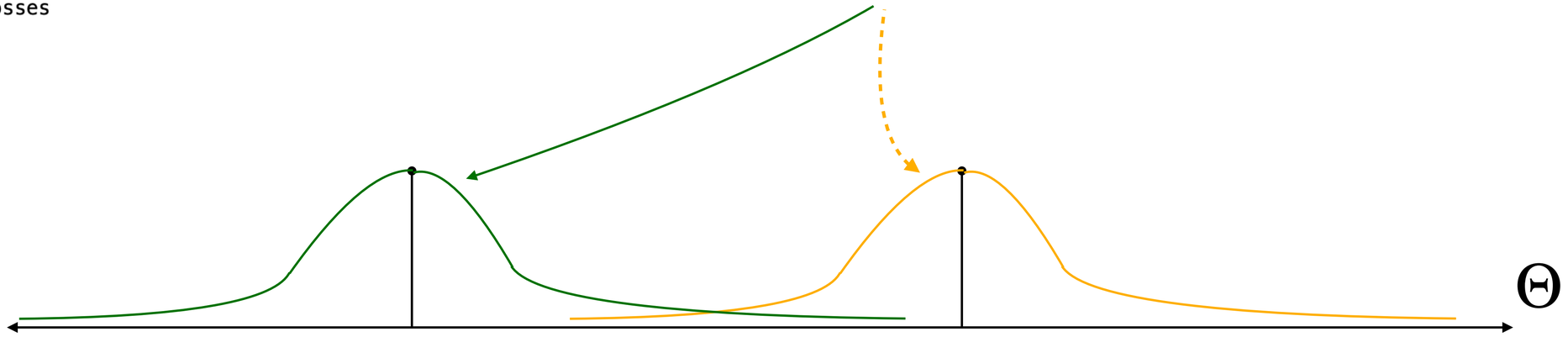
- Increasing σ makes it **harder** to guess if your data was in the training set -> **Increased Privacy**

- But it also means that the **dynamic is noisier** -> **Reduced Utility**

```python
def train_nonprivate(model, epochs=epochs, bs=batch_size, lr=lr):
    opt = optim.SGD(model.parameters(), lr=lr)
    losses = []
    for _ in range(epochs):
        idx = torch.randperm(n)
        for i in range(0, n, bs):
            b = idx[i:i+bs]
            xb, yb = X[b], y[b]
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
            losses.append(loss.item())
    return losses
```

$$\theta \leftarrow \theta - lr \times \left\{ \frac{1}{batch\_size} \left(\sum_{j \in batch} \nabla l(y_j, model(x_j)) + \sigma \mathcal{N}(0, I) \right) \right\}$$

$$\sum_{j \in batch \setminus \{your\_data\}} \nabla l(y_j, model(x_j)) + \mathbf{1}_{your\_data \in batch} \nabla l(y_{your\_data}, model(x_{your\_data})) + \sigma \mathcal{N}(0, I)$$

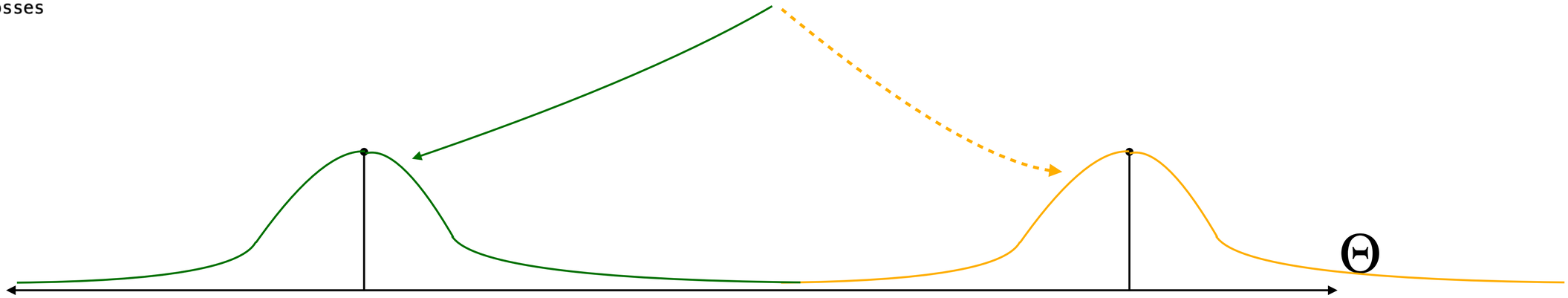


How much the gradients can change

```python
def train_nonprivate(model, epochs=epochs, bs=batch_size, lr=lr):
    opt = optim.SGD(model.parameters(), lr=lr)
    losses = []
    for _ in range(epochs):
        idx = torch.randperm(n)
        for i in range(0, n, bs):
            b = idx[i:i+bs]
            xb, yb = X[b], y[b]
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
            losses.append(loss.item())
    return losses
```

$$\theta \leftarrow \theta - lr \times \left\{ \frac{1}{batch\_size} \left(\sum_{j \in batch} \nabla l(y_j, model(x_j)) + \sigma \mathcal{N}(0,I) \right) \right\}$$

$$\sum_{j \in batch \setminus \{your\_data\}} \nabla l(y_j, model(x_j)) + \mathbf{1}_{your\_data \in batch} \nabla l(y_{your\_data}, model(x_{your\_data})) + \sigma \mathcal{N}(0,I)$$



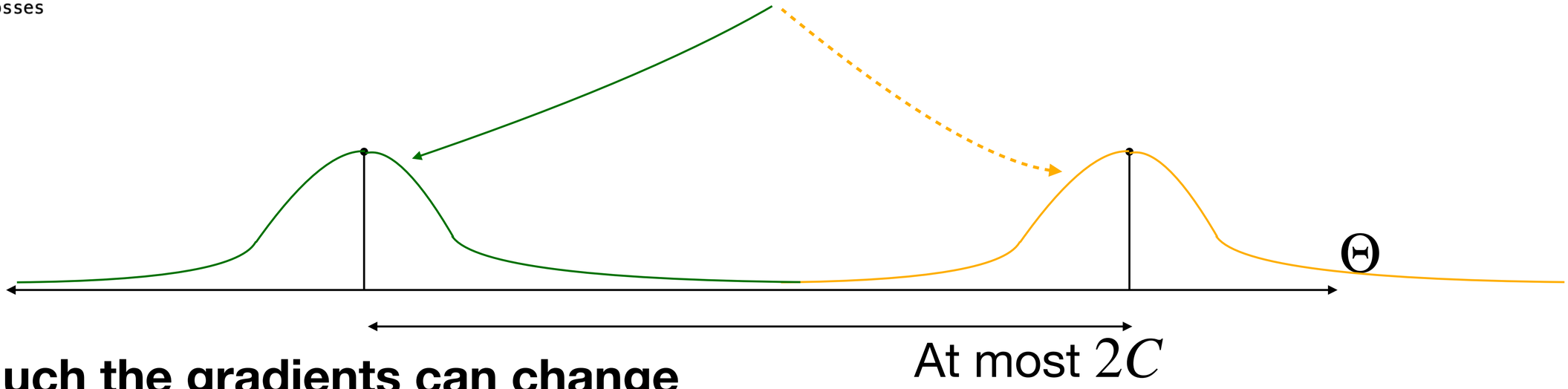
How much the gradients can change

If the **difference between** your gradient and Chuck Norris's **gradient is large**, determining whether your data was included in the training set becomes easy.

```python
def train_nonprivate(model, epochs=epochs, bs=batch_size, lr=lr):
    opt = optim.SGD(model.parameters(), lr=lr)
    losses = []
    for _ in range(epochs):
        idx = torch.randperm(n)
        for i in range(0, n, bs):
            b = idx[i:i+bs]
            xb, yb = X[b], y[b]
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
            losses.append(loss.item())
    return losses
```

$$\theta \leftarrow \theta - lr \times \left\{ \frac{1}{batch\_size} \left(\sum_{j \in batch} clip_C(\nabla l(y_j, model(x_j))) + \sigma \mathcal{N}(0, I) \right) \right\}$$

$$\sum_{j \in batch \setminus \{your\_data\}} clip_C(\nabla l(y_j, model(x_j))) + \mathbf{1}_{your\_data \in batch}\, clip_C(\nabla l(y_{your\_data}, model(x_{your\_data}))) + \sigma \mathcal{N}(0, I)$$



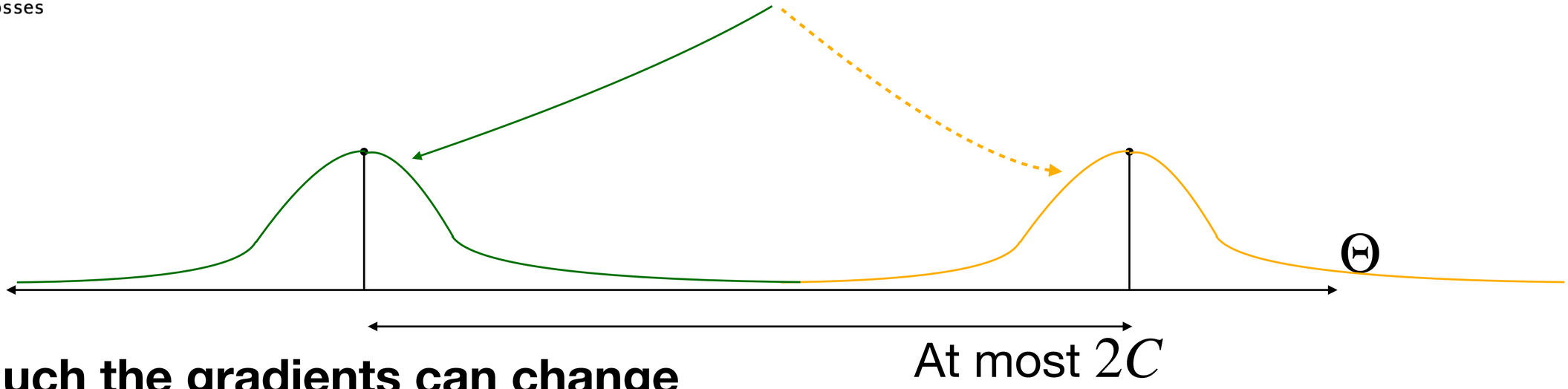
How much the gradients can change

In practice, we make sure that **this difference is not too large** by *clipping* the values of the gradients.

```python
def train_nonprivate(model, epochs=epochs, bs=batch_size, lr=lr):
    opt = optim.SGD(model.parameters(), lr=lr)
    losses = []
    for _ in range(epochs):
        idx = torch.randperm(n)
        for i in range(0, n, bs):
            b = idx[i:i+bs]
            xb, yb = X[b], y[b]
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
            losses.append(loss.item())
    return losses
```

$$\theta \leftarrow \theta - lr \times \left\{ \frac{1}{batch\_size} \left(\sum_{j \in batch} clip_C(\nabla l(y_j, model(x_j))) + \sigma \mathcal{N}(0, I) \right) \right\}$$

$$\sum_{j \in batch \setminus \{your\_data\}} clip_C(\nabla l(y_j, model(x_j))) + \mathbf{1}_{your\_data \in batch}\, clip_C(\nabla l(y_{your\_data}, model(x_{your\_data}))) + \sigma \mathcal{N}(0, I)$$



How much the gradients can change

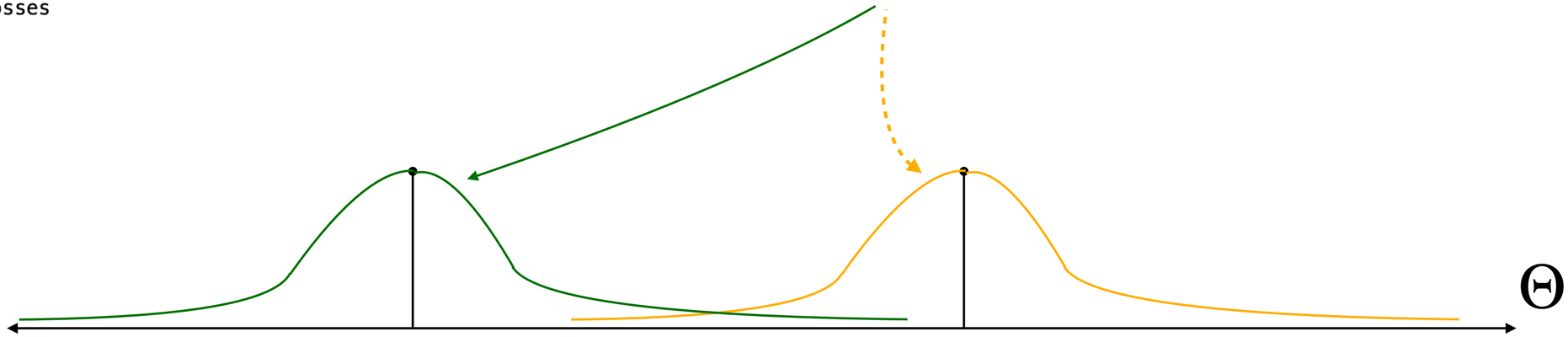
- A lower clipping threshold **reduces the influence of your data** -> **Higher Privacy**.

- However, it **alters the standard SGD dynamics** -> **Reduced Utility**.

```python
def train_nonprivate(model, epochs=epochs, bs=batch_size, lr=lr):
    opt = optim.SGD(model.parameters(), lr=lr)
    losses = []
    for _ in range(epochs):
        idx = torch.randperm(n)
        for i in range(0, n, bs):
            b = idx[i:i+bs]
            xb, yb = X[b], y[b]
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
            losses.append(loss.item())
    return losses
```

$$\theta \leftarrow \theta - lr \times \left\{ \frac{1}{batch\_size} \left(\sum_{j \in batch} \nabla l(y_j, model(x_j)) + \sigma \mathcal{N}(0, I) \right) \right\}$$

$$\sum_{j \in batch \setminus \{your\_data\}} \nabla l(y_j, model(x_j)) + \mathbf{1}_{your\_data \in batch} \nabla l(y_{your\_data}, model(x_{your\_data})) + \sigma \mathcal{N}(0, I)$$

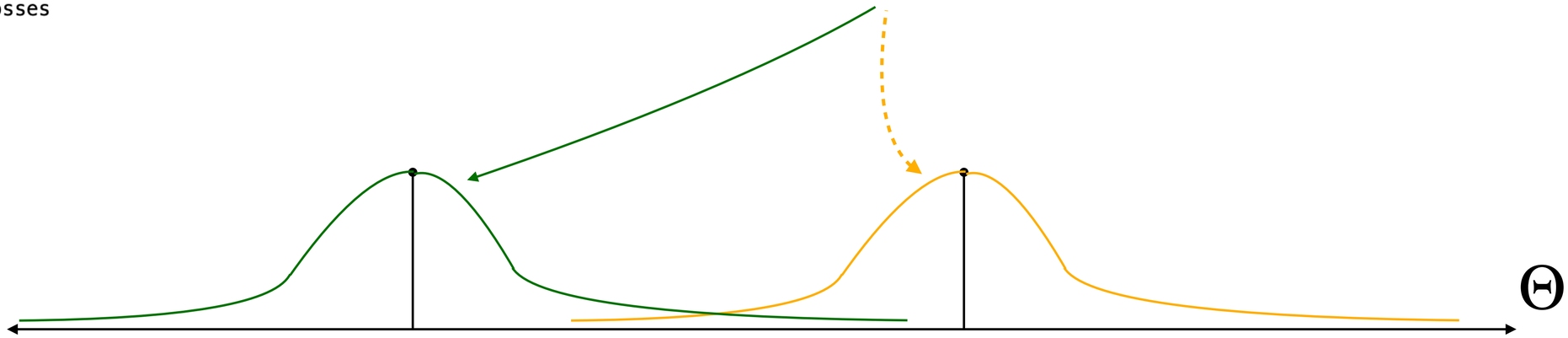


The number of epochs

```python
def train_nonprivate(model, epochs=epochs, bs=batch_size, lr=lr):
    opt = optim.SGD(model.parameters(), lr=lr)
    losses = []
    for _ in range(epochs):
        idx = torch.randperm(n)
        for i in range(0, n, bs):
            b = idx[i:i+bs]
            xb, yb = X[b], y[b]
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
            losses.append(loss.item())
    return losses
```

$$\theta \leftarrow \theta - lr \times \left\{ \frac{1}{batch\_size} \left(\sum_{j \in batch} \nabla l(y_j, model(x_j)) + \sigma \mathcal{N}(0,I) \right) \right\}$$

$$\sum_{j \in batch \setminus \{your\_data\}} \nabla l(y_j, model(x_j)) + \mathbf{1}_{your\_data \in batch} \nabla l(y_{your\_data}, model(x_{your\_data})) + \sigma \mathcal{N}(0,I)$$



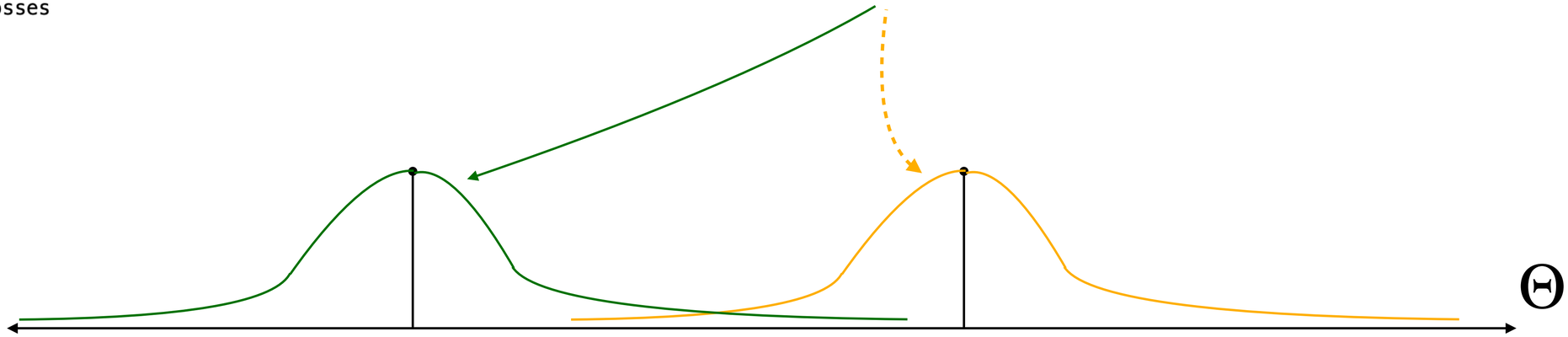
The number of epochs

- **More epochs** -> better optimization and **higher utility**.
- But **more exposure to each user's data** -> **weaker privacy** guarantees.

```python
def train_nonprivate(model, epochs=epochs, bs=batch_size, lr=lr):
    opt = optim.SGD(model.parameters(), lr=lr)
    losses = []
    for _ in range(epochs):
        idx = torch.randperm(n)
        for i in range(0, n, bs):
            b = idx[i:i+bs]
            xb, yb = X[b], y[b]
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
            losses.append(loss.item())
    return losses
```

$$\theta \leftarrow \theta - lr \times \left\{ \frac{1}{batch\_size} \left(\sum_{j \in batch} \nabla l(y_j, model(x_j)) + \sigma \mathcal{N}(0,I) \right) \right\}$$

$$\sum_{j \in batch \setminus \{your\_data\}} \nabla l(y_j, model(x_j)) + \mathbf{1}_{your\_data \in batch} \nabla l(y_{your\_data}, model(x_{your\_data})) + \sigma \mathcal{N}(0,I)$$

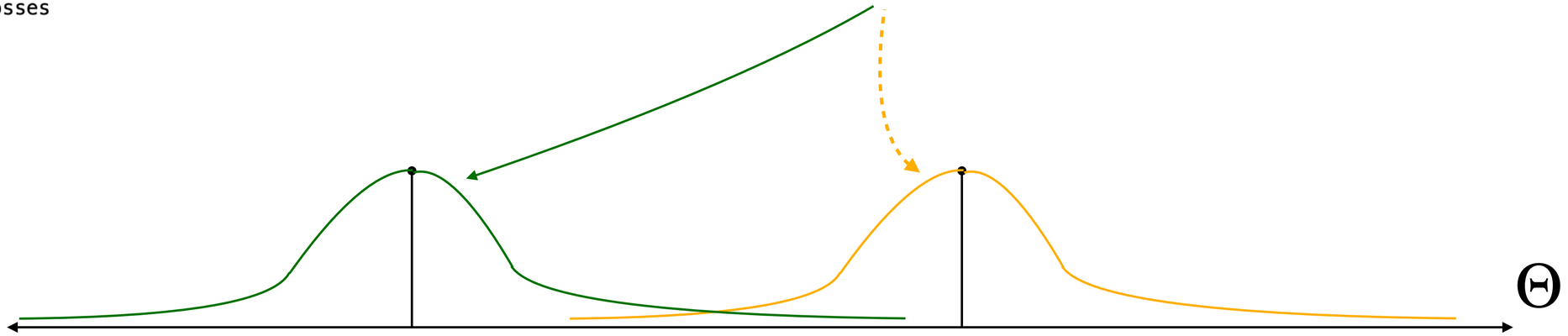


The batching mechanism

```python
def train_nonprivate(model, epochs=epochs, bs=batch_size, lr=lr):
    opt = optim.SGD(model.parameters(), lr=lr)
    losses = []
    for _ in range(epochs):
        idx = torch.randperm(n)
        for i in range(0, n, bs):
            b = idx[i:i+bs]
            xb, yb = X[b], y[b]
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
            losses.append(loss.item())
    return losses
```

$$\theta \leftarrow \theta - lr \times \left\{ \frac{1}{batch\_size} \left(\sum_{j \in batch} \nabla l(y_j, model(x_j)) + \sigma \mathcal{N}(0,I) \right) \right\}$$

$$\sum_{j \in batch \setminus \{your\_data\}} \nabla l(y_j, model(x_j)) + \mathbf{1}_{your\_data \in batch} \nabla l(y_{your\_data}, model(x_{your\_data})) + \sigma \mathcal{N}(0,I)$$



The batching mechanism

Outside of the scope of this presentation, but it is possible to play with “smarter” batching mechanisms to shape the privacy-utility tradeoff.

```
def train_nonprivate(model, epochs=epochs, bs=batch_size, lr=lr):
    opt = optim.SGD(model.parameters(), lr=lr)
    losses = []
    for _ in range(epochs):
        idx = torch.randperm(n
        for i in range(0, n, b
            b = idx[i:i+bs]
            xb, yb = X[b], y[b
            opt.zero_grad()
            loss = loss_fn(mod
            loss.backward()
            opt.step()
            losses.append(loss
    return losses
```

$$\theta \leftarrow \theta - lr \times \left\{ \frac{1}{\text{batch\_size}} \left(\sum \nabla l(y_j, model(x_j)) + \sigma \mathcal{N}(0,I) \right) \right\}$$

$$\ldots del(x_{your\_data})) + \sigma \mathcal{N}(0,I)$$

A preliminary version of this paper appears in the proceedings of the *23rd ACM Conference on Computer and Communications Security (CCS 2016)*. This is a full version.

Deep Learning with Differential Privacy

October 25, 2016

Martín Abadi* Andy Chu* Ian Goodfellow†
H. Brendan McMahan* Ilya Mironov* Kunal Talwar*
Li Zhang*

ABSTRACT

Machine learning techniques based on neural networks are achieving remarkable results in a wide variety of domains. Often, the training of models requires large, representative datasets, which may be crowdsourced and contain sensitive information. The models should not expose private information in these datasets. Addressing this goal, we develop new algorithmic techniques for learning and a refined analysis of privacy costs within the framework of differential privacy.

1. We demonstrate that, by tracking detailed information (higher moments) of the privacy loss, we can obtain much tighter estimates on the overall privacy loss, both asymptotically and empirically.
2. We improve the computational efficiency of differentially private training by introducing new techniques. These techniques include efficient algorithms for computing gradients for individual training examples, sub-

Θ

The batching mechanism

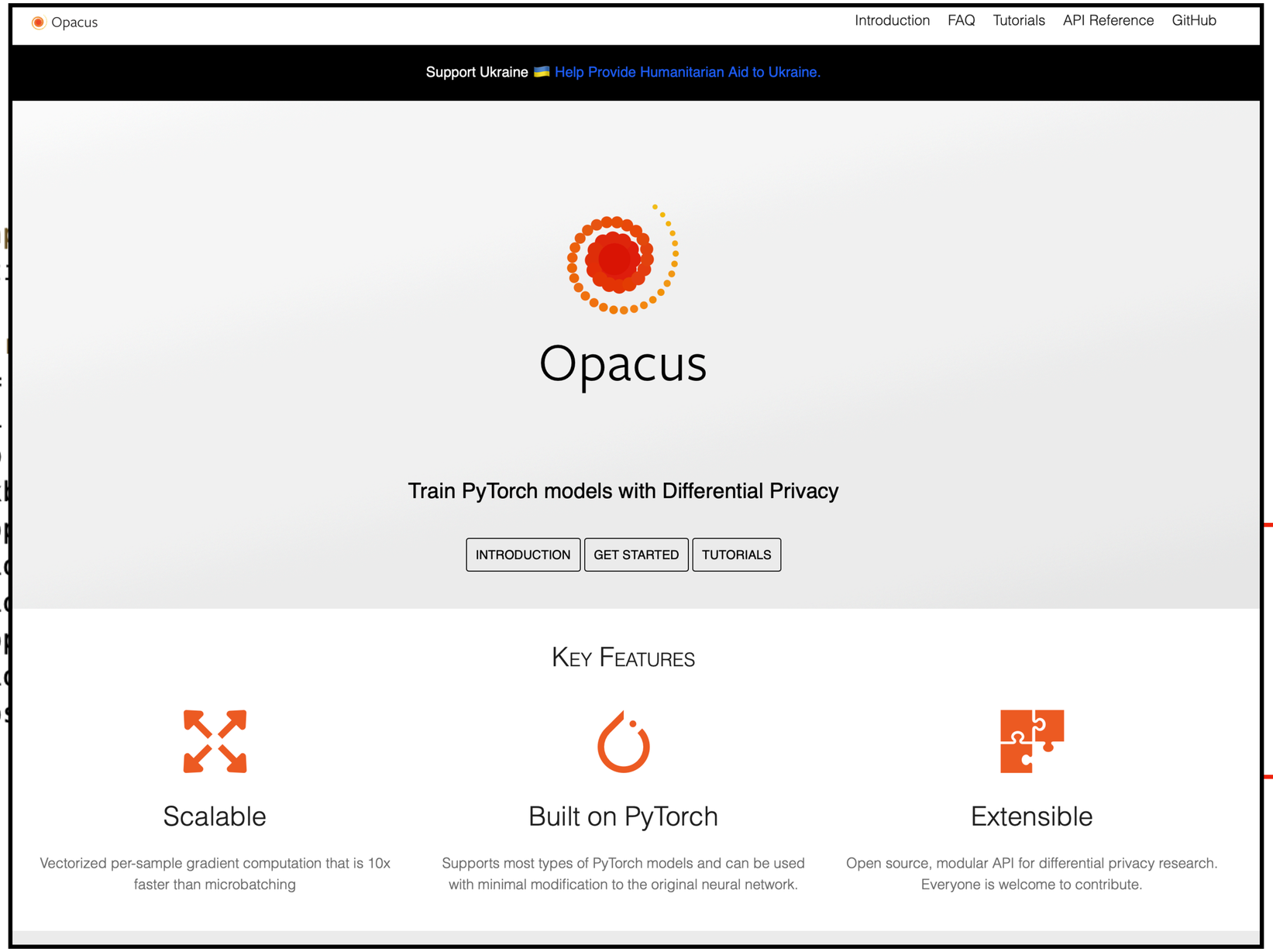
Outside of the scope of this presentation, but it is possible to play with “smarter” batching mechanisms to shape the privacy-utility tradeoff.

ANITI

```python
def train_nonprivate(model, epochs=epochs, bs=batch_size, lr=lr):
    opt = optim.SGD(model.parameters(), lr=lr)
    losses = []
    for _ in range(epochs):
        idx = torch.randperm(n)
        for i in range(0, n, bs):
            b = idx[i:i+bs]
            xb, yb = X[b], y[b]
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
            losses.append(loss.item())
    return losses
```

Clément, enough theory, we just want to make this private already!

```
def train_nonp
    opt = opti
    losses =
    for _ in
        idx =
        for i
            b
            xb
            op
            lo
            lo
            op
            lo
    return los
```



just

```python
def train_nonprivate(model, epochs=epochs, bs=batch_size, lr=lr):
    opt = optim.SGD(model.parameters(), lr=lr)
    losses = []
    for _ in range(epochs):
        idx = torch.randperm(n)
        for i in range(0, n, bs):
            b = idx[i:i+bs]
            xb, yb = X[b], y[b]
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
            losses.append(loss.item())
    return losses
```

```python
def train_opacus(model, epochs=epochs, bs=batch_size, lr=lr, max_grad_norm=1.0, noise_multiplier=1.0, delta=1e-5):
    from opacus import PrivacyEngine

    loader = torch.utils.data.DataLoader(
        torch.utils.data.TensorDataset(X, y),
        batch_size=bs,
        shuffle=True,
    )
    opt = optim.SGD(model.parameters(), lr=lr)
    privacy_engine = PrivacyEngine()

    model, opt, loader = privacy_engine.make_private(
        module=model,
        optimizer=opt,
        data_loader=loader,
        noise_multiplier=noise_multiplier,
        max_grad_norm=max_grad_norm,
    )

    losses = []
    for _ in range(epochs):
        for xb, yb in loader:
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
            losses.append(loss.item())

    eps = privacy_engine.get_epsilon(delta=delta)
    return losses, eps
```

```python
def train_nonprivate(model, epochs=epochs, bs=batch_size, lr=lr):
    opt = optim.SGD(model.parameters(), lr=lr)
    losses = []
    for _ in range(epochs):
        idx = torch.randperm(n)
        for i in range(0, n, bs):
            b = idx[i:i+bs]
            xb, yb = X[b], y[b]
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
            losses.append(loss.item())
    return losses
```

```python
def train_opacus(model, epochs=epochs, bs=batch_size, lr=lr, max_grad_norm=1.0, noise_multiplier=1.0, delta=1e-5):
    from opacus import PrivacyEngine

    loader = torch.utils.data.DataLoader(
        torch.utils.data.TensorDataset(X, y),
        batch_size=bs,
        shuffle=True,
    )
    opt = optim.SGD(model.parameters(), lr=lr)
    privacy_engine = PrivacyEngine()

    model, opt, loader = privacy_engine.make_private(
        module=model,
        optimizer=opt,
        data_loader=loader,
        noise_multiplier=noise_multiplier,
        max_grad_norm=max_grad_norm,
    )

    losses = []
    for _ in range(epochs):
        for xb, yb in loader:
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
            losses.append(loss.item())

    eps = privacy_engine.get_epsilon(delta=delta)
    return losses, eps
```

Extra code to handle Privacy

```python
def train_nonprivate(model, epochs=epochs, bs=batch_size, lr=lr):
    opt = optim.SGD(model.parameters(), lr=lr)
    losses = []
    for _ in range(epochs):
        idx = torch.randperm(n)
        for i in range(0, n, bs):
            b = idx[i:i+bs]
            xb, yb = X[b], y[b]
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
            losses.append(loss.item())
    return losses
```

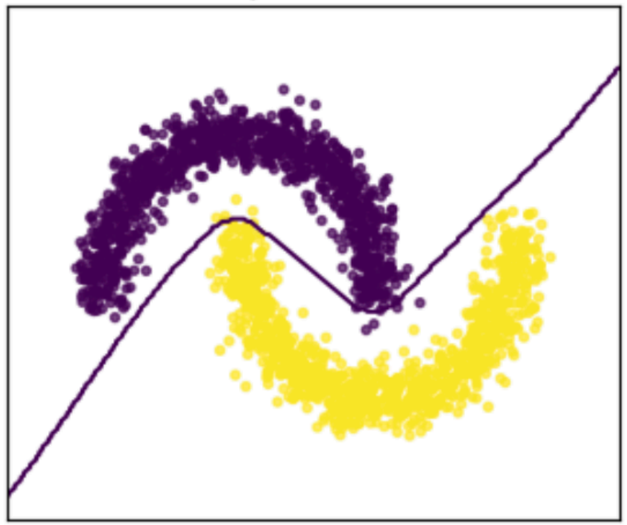
Same training loop

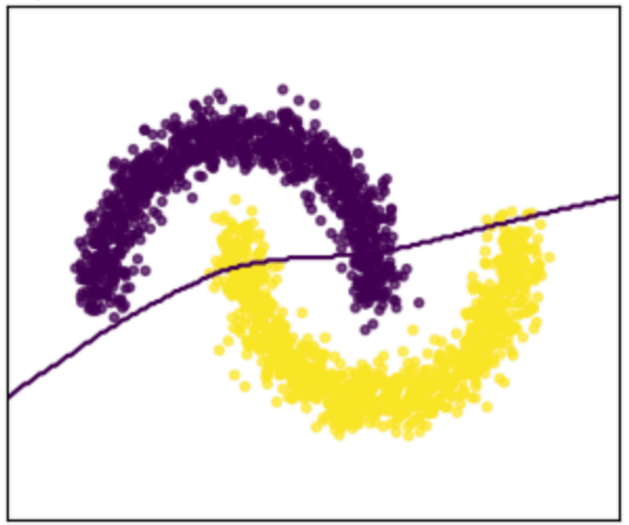
```python
def train_opacus(model, epochs=epochs, bs=batch_size, lr=lr, max_grad_norm=1.0, noise_multiplier=1.0, delta=1e-5):
    from opacus import PrivacyEngine

    loader = torch.utils.data.DataLoader(
        torch.utils.data.TensorDataset(X, y),
        batch_size=bs,
        shuffle=True,
    )
    opt = optim.SGD(model.parameters(), lr=lr)
    privacy_engine = PrivacyEngine()

    model, opt, loader = privacy_engine.make_private(
        module=model,
        optimizer=opt,
        data_loader=loader,
        noise_multiplier=noise_multiplier,
        max_grad_norm=max_grad_norm,
    )

    losses = []
    for _ in range(epochs):
        for xb, yb in loader:
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
            losses.append(loss.item())

    eps = privacy_engine.get_epsilon(delta=delta)
    return losses, eps
```

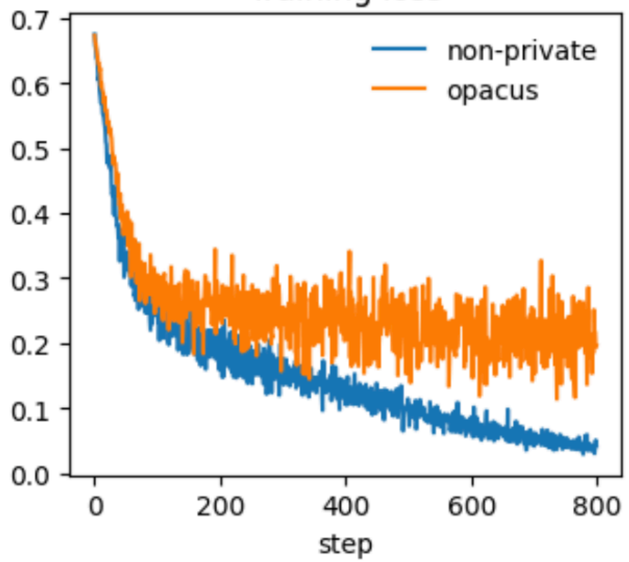
Extra code to handle Privacy

```python
def train_opacus(model, epochs=epochs, bs=batch_size, lr=lr, max_grad_norm=1.0, noise_multiplier=1.0, delta=1e-5):
    from opacus import PrivacyEngine

    loader = torch.utils.data.DataLoader(
        torch.utils.data.TensorDataset(X, y),
        batch_size=bs,
        shuffle=True,
    )
    opt = optim.SGD(model.parameters(), lr=lr)
    privacy_engine = PrivacyEngine()

    model, opt, loader = privacy_engine.make_private(
        module=model,
        optimizer=opt,
        data_loader=loader,
        noise_multiplier=noise_multiplier,
        max_grad_norm=max_grad_norm,
    )

    losses = []
    for _ in range(epochs):
        for xb, yb in loader:
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
            losses.append(loss.item())

    eps = privacy_engine.get_epsilon(delta=delta)
    return losses, eps
```

```
n = 2000
lr = 0.05
batch_size = 256
epochs = 100
max_grad_norm = 1.0
noise_multiplier= 1.0
```



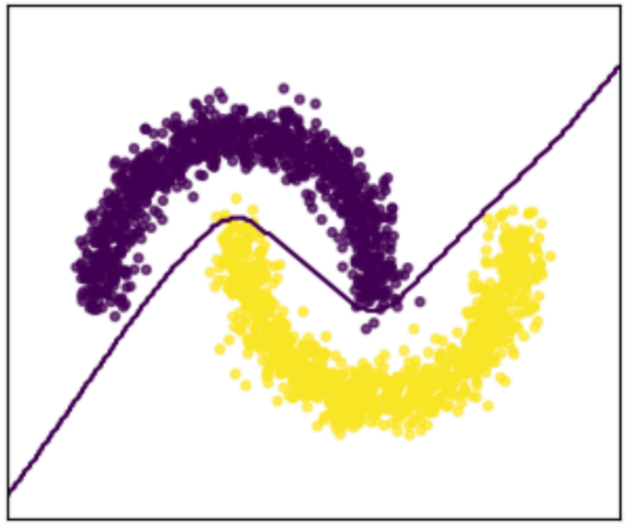


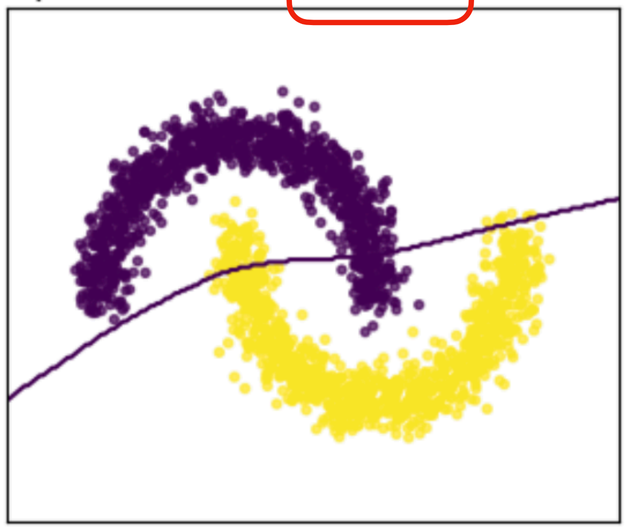


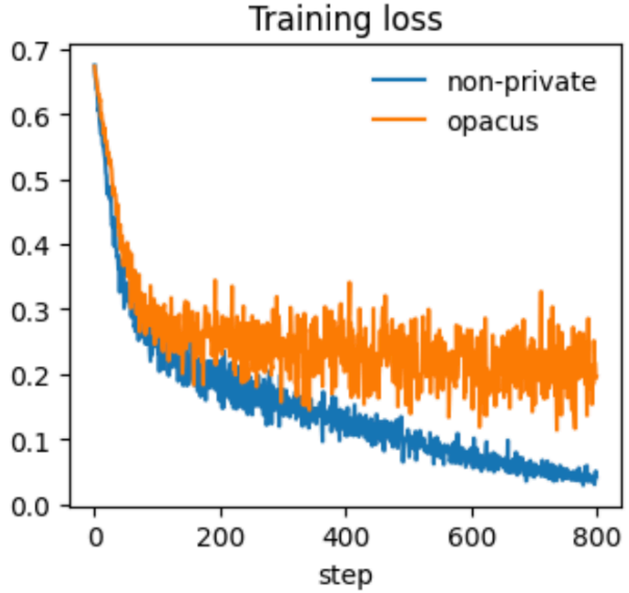
```python
def train_opacus(model, epochs=epochs, bs=batch_size, lr=lr, max_grad_norm=1.0, noise_multiplier=1.0, delta=1e-5):
    from opacus import PrivacyEngine

    loader = torch.utils.data.DataLoader(
        torch.utils.data.TensorDataset(X, y),
        batch_size=bs,
        shuffle=True,
    )
    opt = optim.SGD(model.parameters(), lr=lr)
    privacy_engine = PrivacyEngine()

    model, opt, loader = privacy_engine.make_private(
        module=model,
        optimizer=opt,
        data_loader=loader,
        noise_multiplier=noise_multiplier,
        max_grad_norm=max_grad_norm,
    )

    losses = []
    for _ in range(epochs):
        for xb, yb in loader:
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
            losses.append(loss.item())

    eps = privacy_engine.get_epsilon(delta=delta)
    return losses, eps
```

Privacy Budget: The smaller, the more private.

```
n = 2000
lr = 0.05
batch_size = 256
epochs = 100
max_grad_norm = 1.0
noise_multiplier= 1.0
```

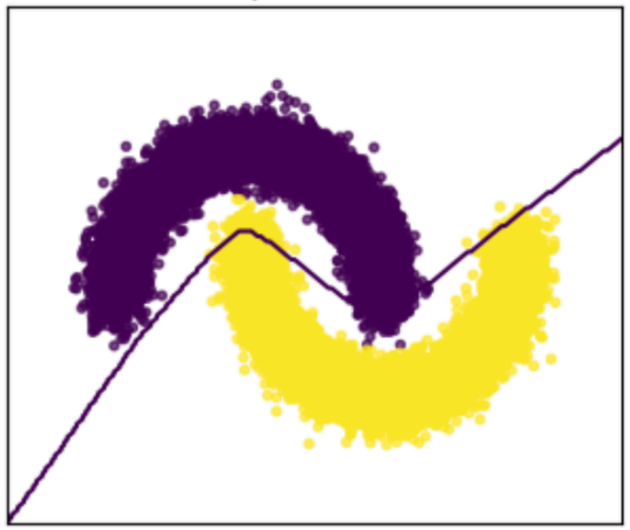


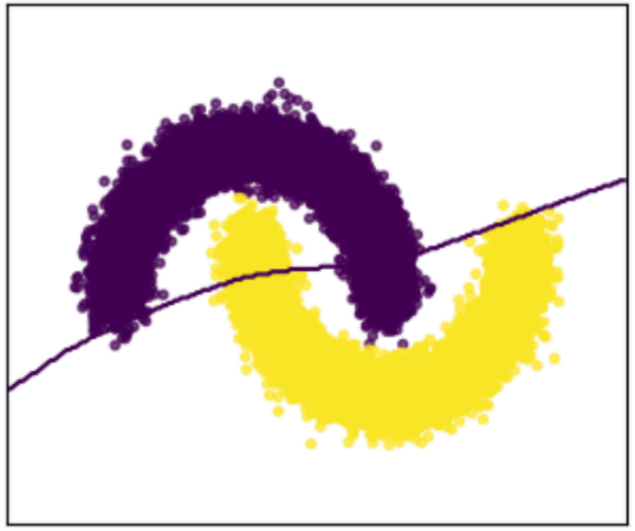


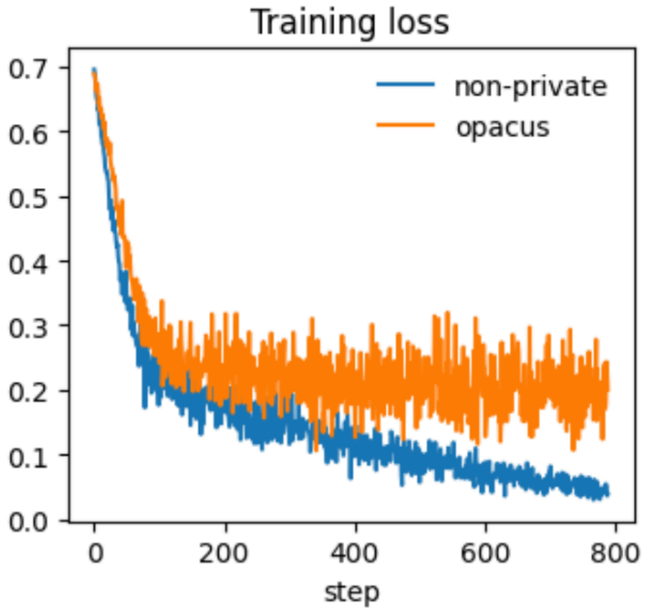


```python
def train_opacus(model, epochs=epochs, bs=batch_size, lr=lr, max_grad_norm=1.0, noise_multiplier=1.0, delta=1e-5):
    from opacus import PrivacyEngine

    loader = torch.utils.data.DataLoader(
        torch.utils.data.TensorDataset(X, y),
        batch_size=bs,
        shuffle=True,
    )
    opt = optim.SGD(model.parameters(), lr=lr)
    privacy_engine = PrivacyEngine()

    model, opt, loader = privacy_engine.make_private(
        module=model,
        optimizer=opt,
        data_loader=loader,
        noise_multiplier=noise_multiplier,
        max_grad_norm=max_grad_norm,
    )

    losses = []
    for _ in range(epochs):
        for xb, yb in loader:
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
            losses.append(loss.item())

    eps = privacy_engine.get_epsilon(delta=delta)
    return losses, eps
```

```
n = 20000
lr = 0.05
batch_size = 256
epochs = 10
max_grad_norm = 1.0
noise_multiplier= 1.0
```

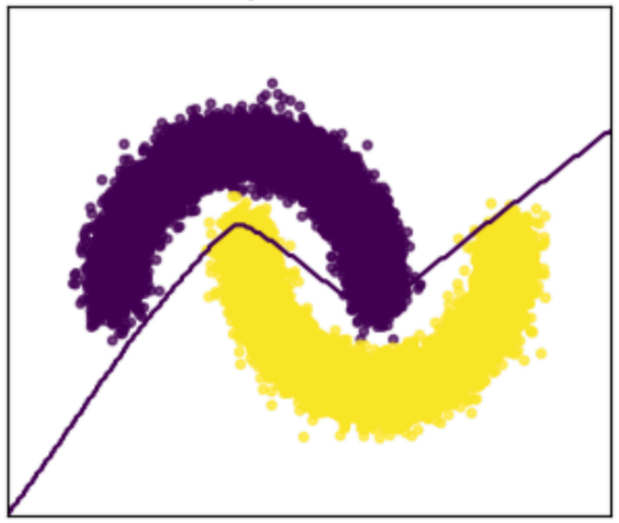


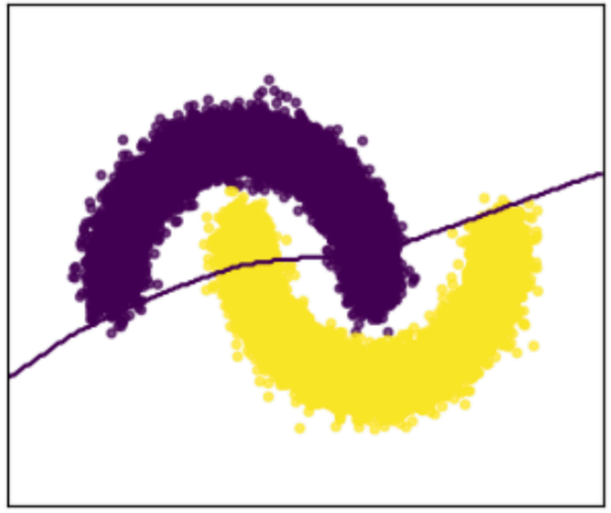


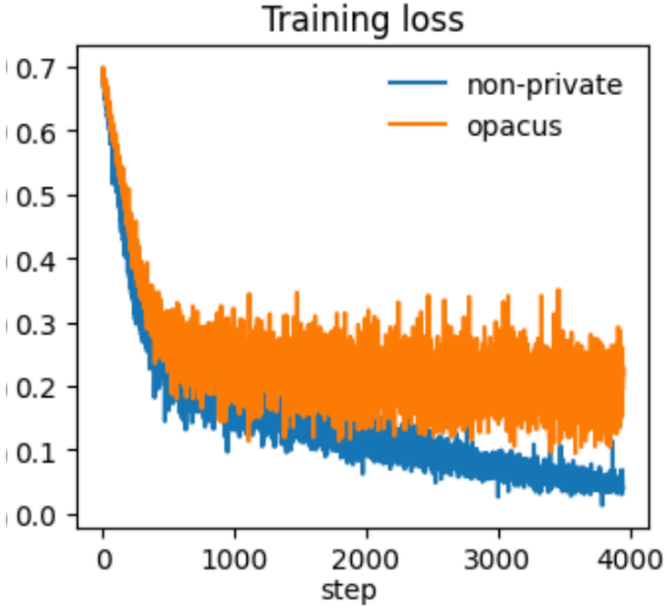


```python
def train_opacus(model, epochs=epochs, bs=batch_size, lr=lr, max_grad_norm=1.0, noise_multiplier=1.0, delta=1e-5):
    from opacus import PrivacyEngine

    loader = torch.utils.data.DataLoader(
        torch.utils.data.TensorDataset(X, y),
        batch_size=bs,
        shuffle=True,
    )
    opt = optim.SGD(model.parameters(), lr=lr)
    privacy_engine = PrivacyEngine()

    model, opt, loader = privacy_engine.make_private(
        module=model,
        optimizer=opt,
        data_loader=loader,
        noise_multiplier=noise_multiplier,
        max_grad_norm=max_grad_norm,
    )

    losses = []
    for _ in range(epochs):
        for xb, yb in loader:
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
            losses.append(loss.item())

    eps = privacy_engine.get_epsilon(delta=delta)
    return losses, eps
```

```
n = 20000
lr = 0.01
batch_size = 256
epochs = 50
max_grad_norm = 1.0
noise_multiplier= 1.0
```

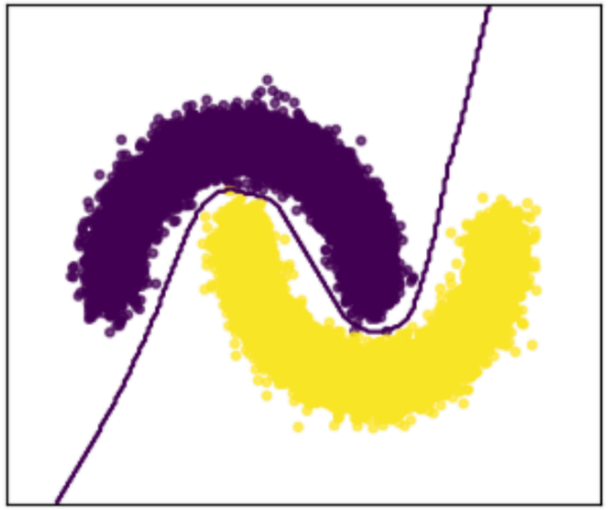


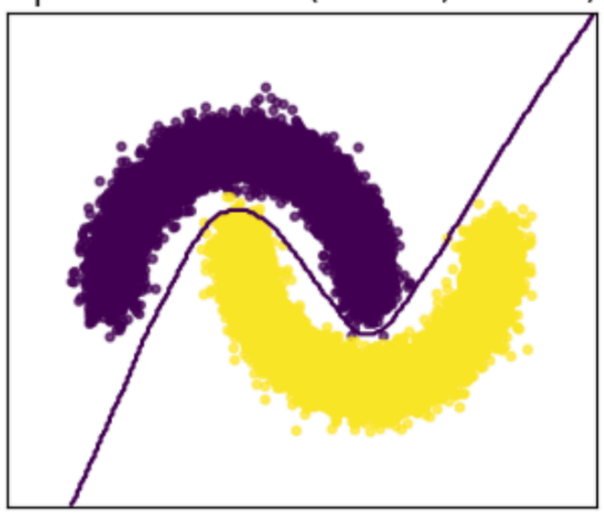


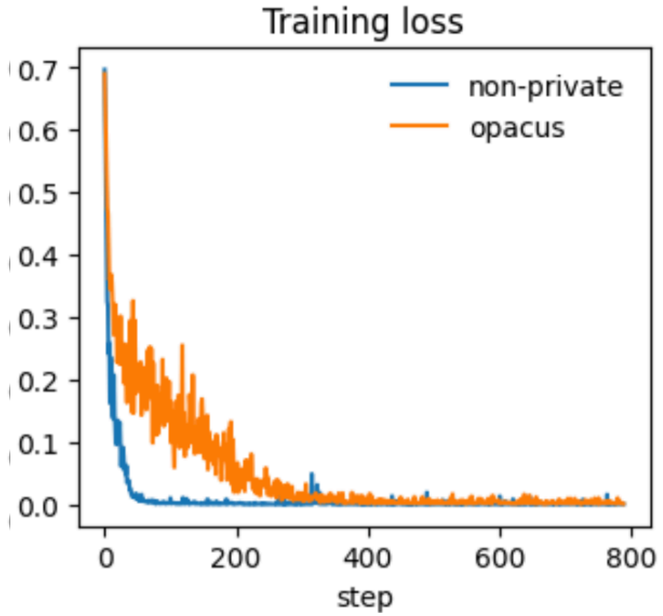


```python
def train_opacus(model, epochs=epochs, bs=batch_size, lr=lr, max_grad_norm=1.0, noise_multiplier=1.0, delta=1e-5):
    from opacus import PrivacyEngine

    loader = torch.utils.data.DataLoader(
        torch.utils.data.TensorDataset(X, y),
        batch_size=bs,
        shuffle=True,
    )
    opt = optim.Adam(model.parameters(), lr=lr)
    privacy_engine = PrivacyEngine()

    model, opt, loader = privacy_engine.make_private(
        module=model,
        optimizer=opt,
        data_loader=loader,
        noise_multiplier=noise_multiplier,
        max_grad_norm=max_grad_norm,
    )

    losses = []
    for _ in range(epochs):
        for xb, yb in loader:
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
            losses.append(loss.item())

    eps = privacy_engine.get_epsilon(delta=delta)
    return losses, eps
```

```
n = 20000
lr = 0.01
batch_size = 256
epochs = 10
max_grad_norm = 1.0
noise_multiplier= 2.5
```







Going further

ANITI

Going further

Opacus

Introduction FAQ Tutorials API Reference GitHub

Tutorials

Overview

Using Opacus

Building text classifier with Fast Gradient Clipping DP-SGD

Building image classifier with Differential Privacy

Training a differentially private LSTM model for name classification

Deep dive into advanced features of Opacus

Guide to Module Validator and Fixer

Guide to grad samplers

Training on multiple GPUs with DistributedDataParallel

Tutorials

This is the tutorials page. Navigate the sidebar to find various tutorials.

External Blog Posts

Introducing Opacus, by Meta AI

Differential Privacy Blog Post Series

1. DP-SGD Algorithm Explained
2. Efficient Per-Sample Gradient Computation in Opacus
3. Efficient Per-Sample Gradient Computation for More Layers in Opacus
4. Enabling Fast Gradient Clipping and Ghost Clipping in Opacus

Videos*

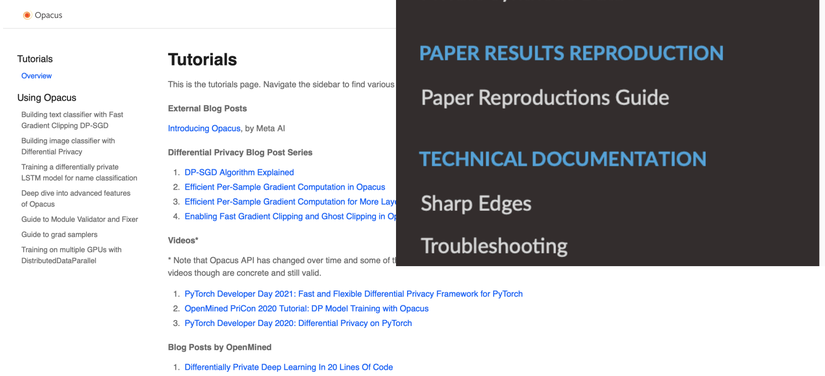
* Note that Opacus API has changed over time and some of the code samples and demos in the videos may not work. The concepts presented in the videos though are concrete and still valid.

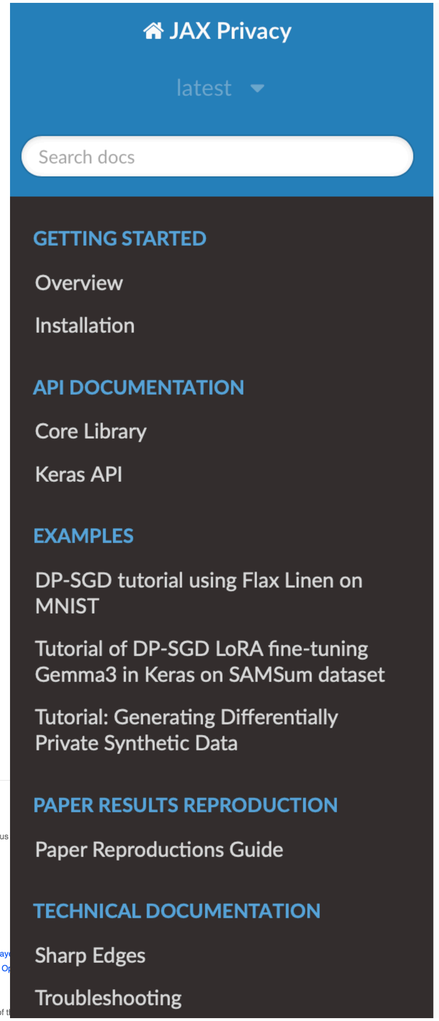
1. PyTorch Developer Day 2021: Fast and Flexible Differential Privacy Framework for PyTorch
2. OpenMined PriCon 2020 Tutorial: DP Model Training with Opacus
3. PyTorch Developer Day 2020: Differential Privacy on PyTorch

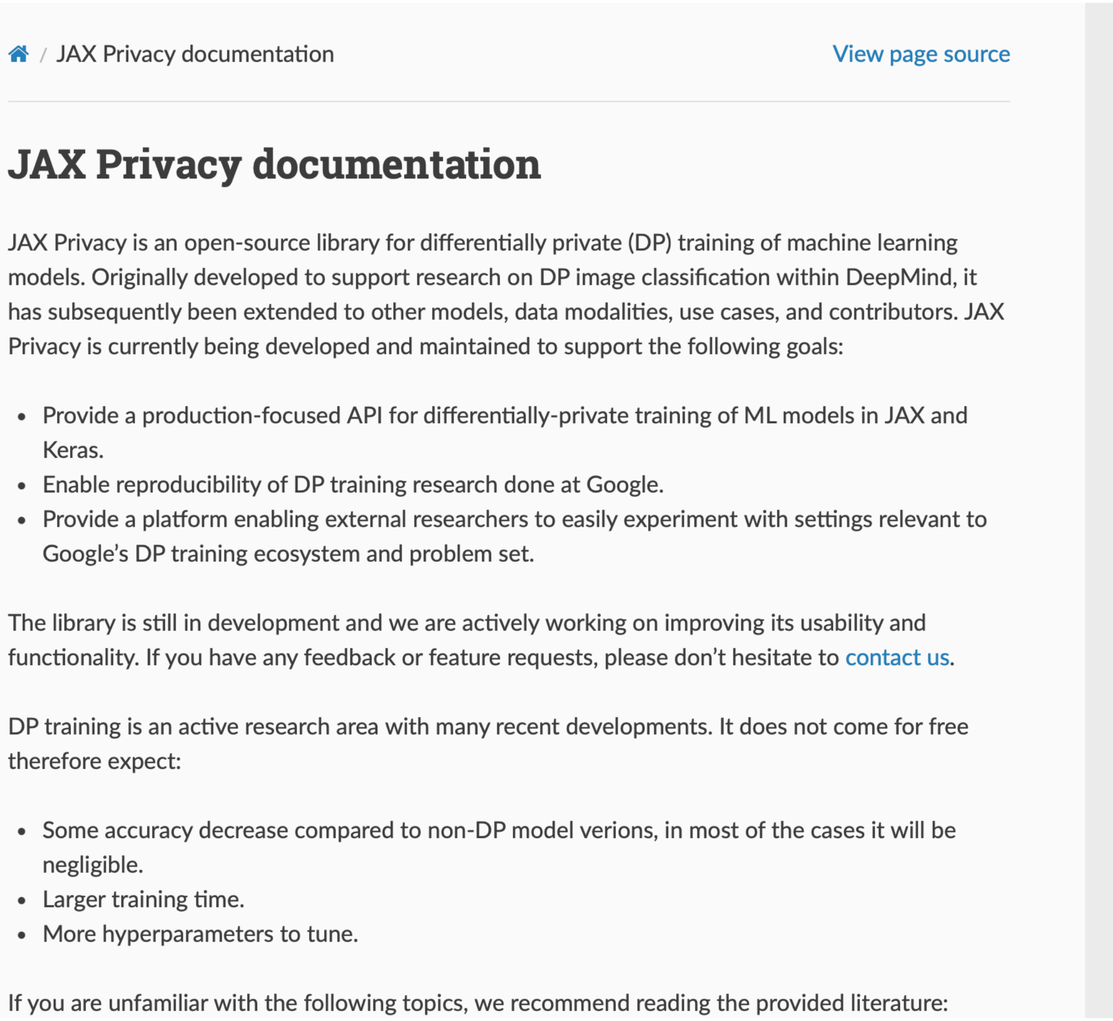
Blog Posts by OpenMined

1. Differentially Private Deep Learning In 20 Lines Of Code

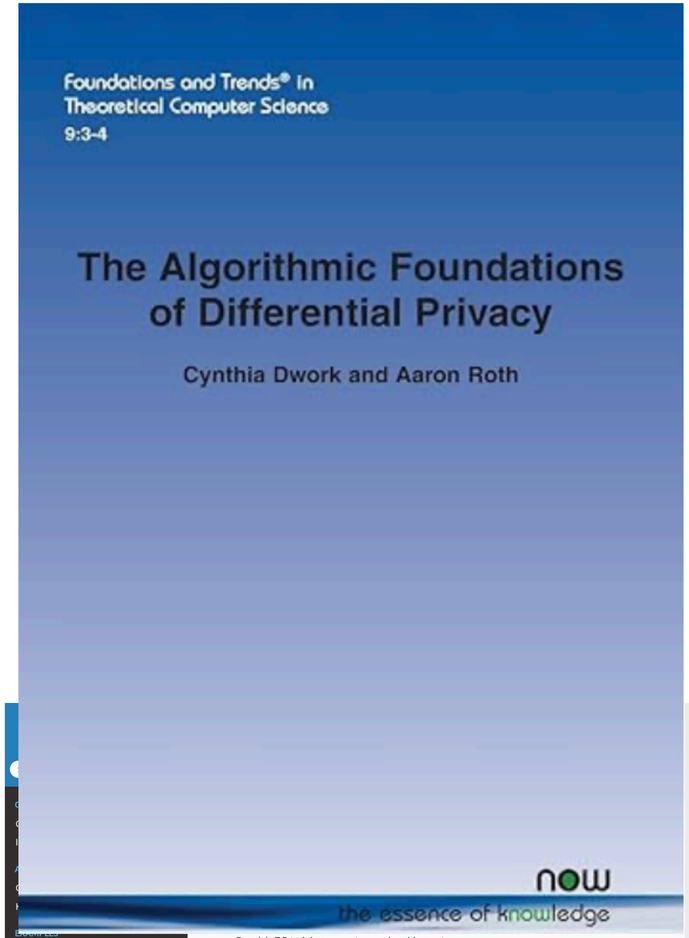
Going further







Going further



Opacus

Introduction FAQ Tutorials API Reference GitHub

Tutorials

Overview

Using Opacus

Building text classifier with Fast Gradient Clipping DP-SGD

Building image classifier with Differential Privacy

Training a differentially private LSTM model for name classification

Deep dive into advanced features of Opacus

Guide to Module Validator and Fixer

Guide to grad samplers

Training on multiple GPUs with DistributedDataParallel

Tutorials

This is the tutorials page. Navigate the sidebar to find various tutorials.

External Blog Posts

Introducing Opacus, by Meta AI

Differential Privacy Blog Post Series

1. DP-SGD Algorithm Explained
2. Efficient Per-Sample Gradient Computation in Opacus
3. Efficient Per-Sample Gradient Computation for More Layers in Opacus
4. Enabling Fast Gradient Clipping and Ghost Clipping in Opacus

Videos*

* Note that Opacus API has changed over time and some of the code samples and demos in the videos may not work. The concepts presented in the videos though are concrete and still valid.

1. PyTorch Developer Day 2021: Fast and Flexible Differential Privacy Framework for PyTorch
2. OpenMined PriCon 2020 Tutorial: DP Model Training with Opacus
3. PyTorch Developer Day 2020: Differential Privacy on PyTorch

Blog Posts by OpenMined

1. Differentially Private Deep Learning In 20 Lines Of Code

DP-SGD tutorial using Flax Linen on MNIST

Tutorial of DP-SGD LoRA fine-tuning Gemma3 in Keras on SAMSum dataset

Tutorial: Generating Differentially Private Synthetic Data

PAPER RESULTS REPRODUCTION

Paper Reproductions Guide

TECHNICAL DOCUMENTATION

Sharp Edges

Troubleshooting

Google's DP training ecosystem and problem set.

The library is still in development and we are actively working on improving its usability and functionality. If you have any feedback or feature requests, please don't hesitate to contact us.

DP training is an active research area with many recent developments. It does not come for free therefore expect:

- Some accuracy decrease compared to non-DP model verions, in most of the cases it will be negligible.
- Larger training time.
- More hyperparameters to tune.

If you are unfamiliar with the following topics, we recommend reading the provided literature:

Going further

Gaussian Differential Privacy

Jinshuo Dong* Aaron Roth† Weijie J. Su‡

May 24, 2019

Abstract

In the past decade, differential privacy has seen remarkable success as a rigorous and practical formalization of data privacy. This privacy definition and its divergence based relaxations, however, have several acknowledged weaknesses, either in handling composition of private algorithms or in analyzing important primitives like privacy amplification by subsampling. Inspired

Opacus

Tutorials

Overview

Using Opacus

Building text classifier with Fast Gradient Clipping DP-SGD

Building image classifier with Differential Privacy

Training a differentially private LSTM model for name classification

Deep dive into advanced features of Opacus

Guide to Module Validator and Fixer

Guide to grad samplers

Training on multiple GPUs with DistributedDataParallel

Tutorials

This is the tutorials page. Navigate the sidebar to find various tutorial

External Blog Posts

Introducing Opacus, by Meta AI

Differential Privacy Blog Post Series

1. DP-SGD Algorithm Explained
2. Efficient Per-Sample Gradient Computation in Opacus
3. Efficient Per-Sample Gradient Computation for More Layers in Opacus
4. Enabling Fast Gradient Clipping and Ghost Clipping in Opacus

Videos*

* Note that Opacus API has changed over time and some of the code samples and demos in the videos may not work. The concepts presented in the videos though are concrete and still valid.

1. PyTorch Developer Day 2021: Fast and Flexible Differential Privacy Framework for PyTorch
2. OpenMined PriCon 2020 Tutorial: DP Model Training with Opacus
3. PyTorch Developer Day 2020: Differential Privacy on PyTorch

Blog Posts by OpenMined

1. Differentially Private Deep Learning In 20 Lines Of Code

Core Library

Keras API

EXAMPLES

DP-SGD tutorial using Flax Linen on MNIST

Tutorial of DP-SGD LoRA fine-tuning Gemma3 in Keras on SAMSum dataset

Tutorial: Generating Differentially Private Synthetic Data

PAPER RESULTS REPRODUCTION

Paper Reproductions Guide

TECHNICAL DOCUMENTATION

Sharp Edges

Troubleshooting

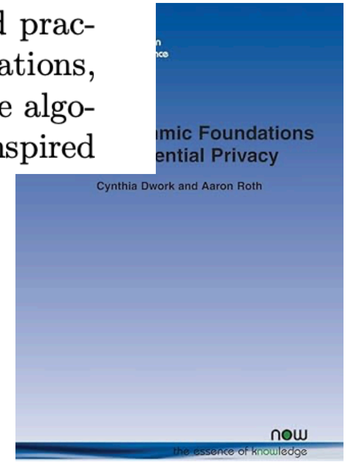
- Provide a production-focused API for differentially-private training of ML models in JAX and Keras.
- Enable reproducibility of DP training research done at Google.
- Provide a platform enabling external researchers to easily experiment with settings relevant to Google's DP training ecosystem and problem set.

The library is still in development and we are actively working on improving its usability and functionality. If you have any feedback or feature requests, please don't hesitate to contact us.

DP training is an active research area with many recent developments. It does not come for free therefore expect:

- Some accuracy decrease compared to non-DP model verions, in most of the cases it will be negligible.
- Larger training time.
- More hyperparameters to tune.

If you are unfamiliar with the following topics, we recommend reading the provided literature:



ANITI

Thank you!

ANITI

Thank you!

Contact: clement.lalanne@math.univ-toulouse.fr