Introduction to Pytorch

CPEN 455 Tutorial 5

Muchen Li & Sadegh Mahdavi

Last Time

- Autograd
  - Disable Autograd
  - Check gradient
  - Demo
- Build your model
  - __init__ & forward function
  - Linear Layer: a case study

This Time

- Model Optimization
  - Loss function
  - Optimizer
- Training & Testing Loop
  - What to expect
  - Save & Load the model
  - Test the model
- Hyper Parameters Tuning

5. Model Optimization: Loss

Loss function measures the degree of dissimilarity of obtained result from our network output to the target value, and it is the loss function that we want to minimize during training.

5. Model Optimization: Loss

MSE Loss:

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right)^2$$

Negative Log Likelihood for N class Classification:

$$-\sum_{c=1}^{N} y_c log(p_c)$$

yc = flag(y==c)

[1] Check out Various Loss functions here: https://pytorch.org/docs/stable/nn.html#loss-functions

[2] Read More About Cross Entropy Loss: https://wandb.ai/sauravmaheshkar/cross-entropy/reports/What-Is-Cross-Entropy-Loss-A-Tutorial-With-Code--VmlldzoxMDA5NTMx

[3] https://en.wikipedia.org/wiki/Cross-entropy

5. Model Optimization: Optimizer

Optimization is the process of adjusting model parameters to reduce model error in each training step. **Optimization algorithms** define how this process is performed (in this example we use Stochastic Gradient Descent). All optimization logic is encapsulated in the optimizer object.

```
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

[4] Check out different optimizers implemented in pytorch: https://pytorch.org/docs/stable/optim.html

5. Model Optimization: Optimizer

Optimization is the process of adjusting model parameters to reduce model error in each training step. **Optimization algorithms** define how this process is performed (in this example we use Stochastic Gradient Descent). All optimization logic is encapsulated in the optimizer object.

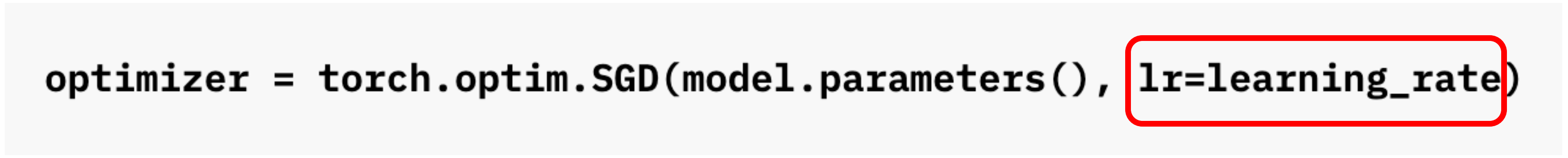
```
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

All parameters within model, only parameter with requires_grad = True will be updated

[4] Check out different optimizers implemented in pytorch: https://pytorch.org/docs/stable/optim.html

5. Model Optimization: Optimizer

Optimization is the process of adjusting model parameters to reduce model error in each training step. **Optimization algorithms** define how this process is performed (in this example we use Stochastic Gradient Descent). All optimization logic is encapsulated in the optimizer object.



```
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

Learning Rate defines the magnitude of a parameter is updated each time.

[4] Check out different optimizers implemented in pytorch: https://pytorch.org/docs/stable/optim.html

5. Model Optimization: Optimizer

Inside the training loop, optimization happens in three steps:

- **optimizer.zero_grad()**: reset the gradients of model parameters. Gradients by default add up; to prevent double-counting, we explicitly zero them at each iteration.

[4] Check out different optimizers implemented in pytorch: https://pytorch.org/docs/stable/optim.html

5. Model Optimization: Optimizer

Inside the training loop, optimization happens in three steps:

- **optimizer.zero_grad()**: reset the gradients of model parameters. Gradients by default add up; to prevent double-counting, we explicitly zero them at each iteration.

- **loss.backward()**: Backpropagate the prediction loss. PyTorch deposits the gradients of the loss w.r.t. each parameter.

[4] Check out different optimizers implemented in pytorch: https://pytorch.org/docs/stable/optim.html

5. Model Optimization: Optimizer

Inside the training loop, optimization happens in three steps:

- **optimizer.zero_grad()**: reset the gradients of model parameters. Gradients by default add up; to prevent double-counting, we explicitly zero them at each iteration.
- **loss.backward()**: Backpropagate the prediction loss. PyTorch deposits the gradients of the loss w.r.t. each parameter.
- **optimizer.step()**: Adjust the parameters by the gradients collected in the backward pass.

[4] Check out different optimizers implemented in pytorch: https://pytorch.org/docs/stable/optim.html

5. Model Optimization: Optimizer

Inside the training loop, optimization happens in three steps:

- **optimizer.zero_grad()**: reset the gradients of model parameters. Gradients by default add up; to prevent double-counting, we explicitly zero them at each iteration.

- **loss.backward()**: Backpropagate the prediction loss. PyTorch deposits the gradients of the loss w.r.t. each parameter.

- **optimizer.step()**: Adjust the parameters by the gradients collected in the backward pass.

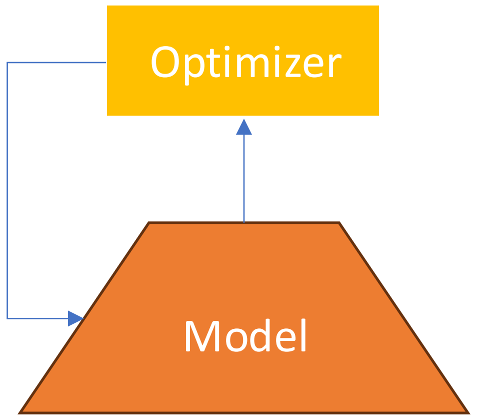
Popular optimizer includes: SGD, Adam, AdamW

[4] Check out different optimizers implemented in pytorch: https://pytorch.org/docs/stable/optim.html

6. Training & Testing

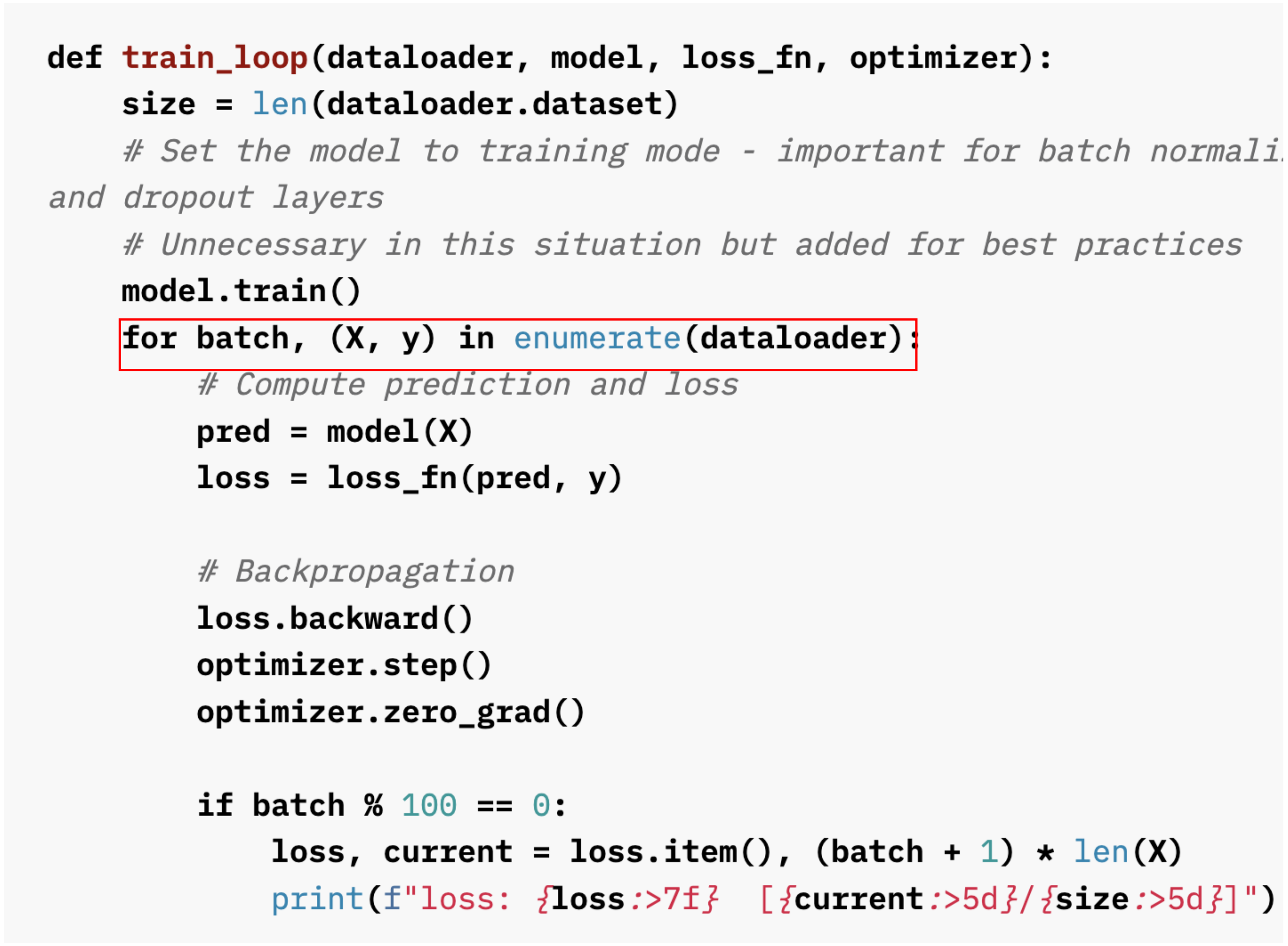
Let's Look at what will be done in one epoch of training:

```
For X, Y_gt in TrainLoader:
    Y_pred = Model(X)
    Loss = LossFunction(Y_pred, Y_gt)
    Loss.backward()
    Optimizer.step()
    Optimizer.zero_grad()
```



6. Training & Testing

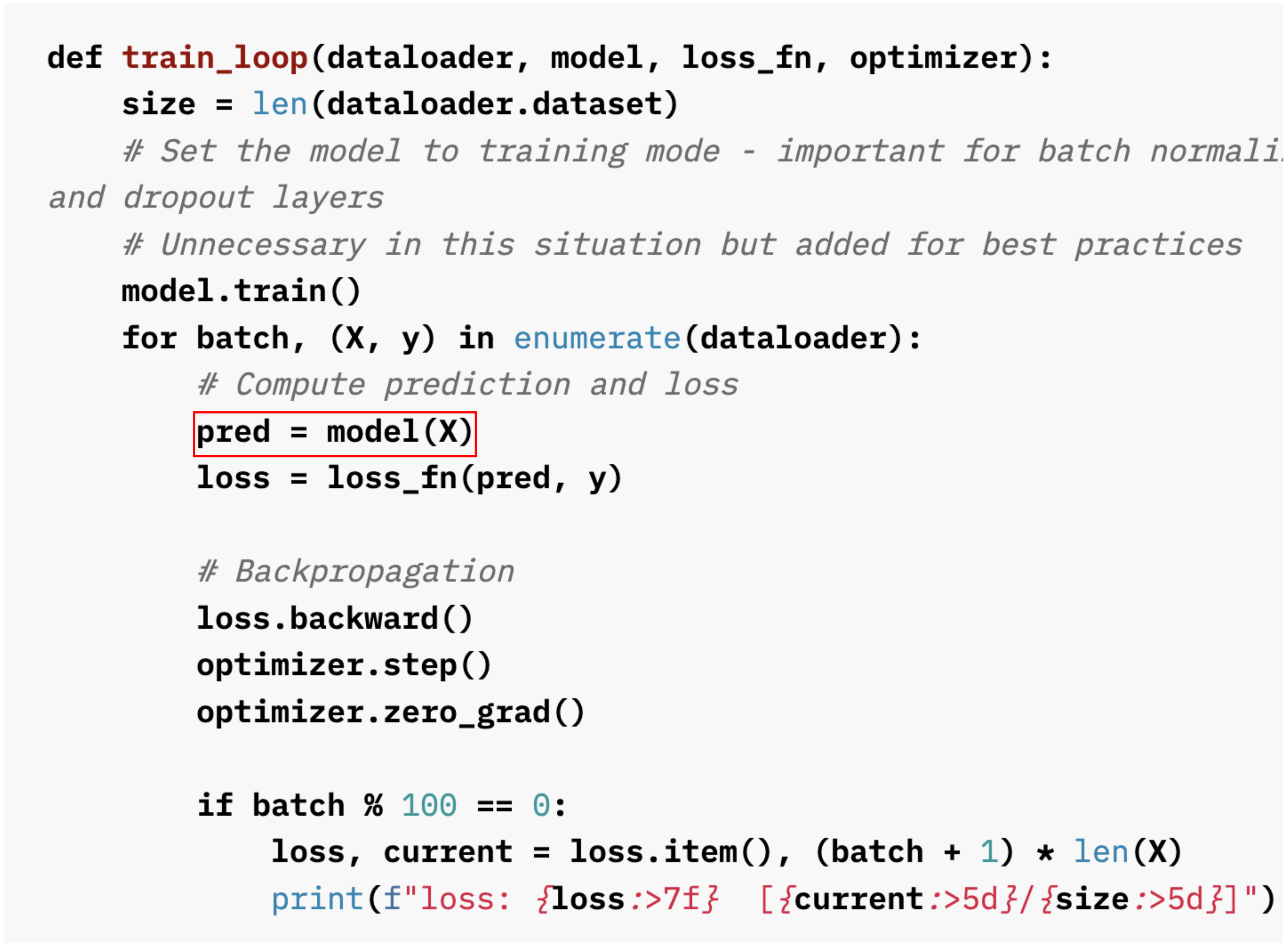
```
For X, Y_gt in TrainLoader:
    Y_pred = Model(X)
    Loss = LossFunction(Y_pred, Y_gt)
    Loss.backward()
    Optimizer.step()
    Optimizer.zero_grad()
```



```
def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    # Set the model to training mode - important for batch normali
and dropout layers
    # Unnecessary in this situation but added for best practices
    model.train()
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction and loss
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

        if batch % 100 == 0:
            loss, current = loss.item(), (batch + 1) * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
```

6. Training & Testing

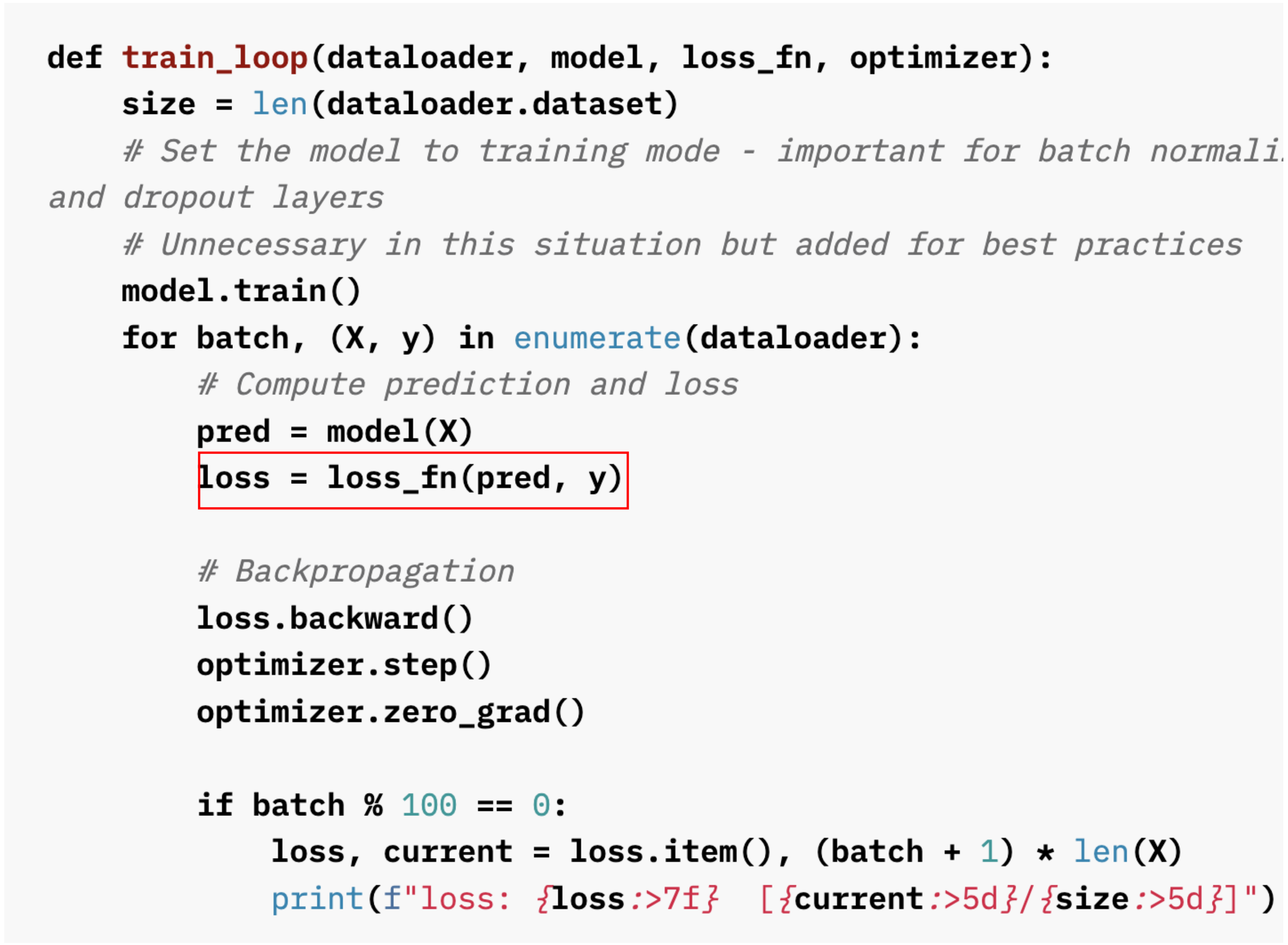
```
For X, Y_gt in TrainLoader:
    Y_pred = Model(X)
    Loss = LossFunction(Y_pred, Y_gt)
    Loss.backward()
    Optimizer.step()
    Optimizer.zero_grad()
```



```python
def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    # Set the model to training mode - important for batch normali
and dropout layers
    # Unnecessary in this situation but added for best practices
    model.train()
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction and loss
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

        if batch % 100 == 0:
            loss, current = loss.item(), (batch + 1) * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
```

6. Training & Testing

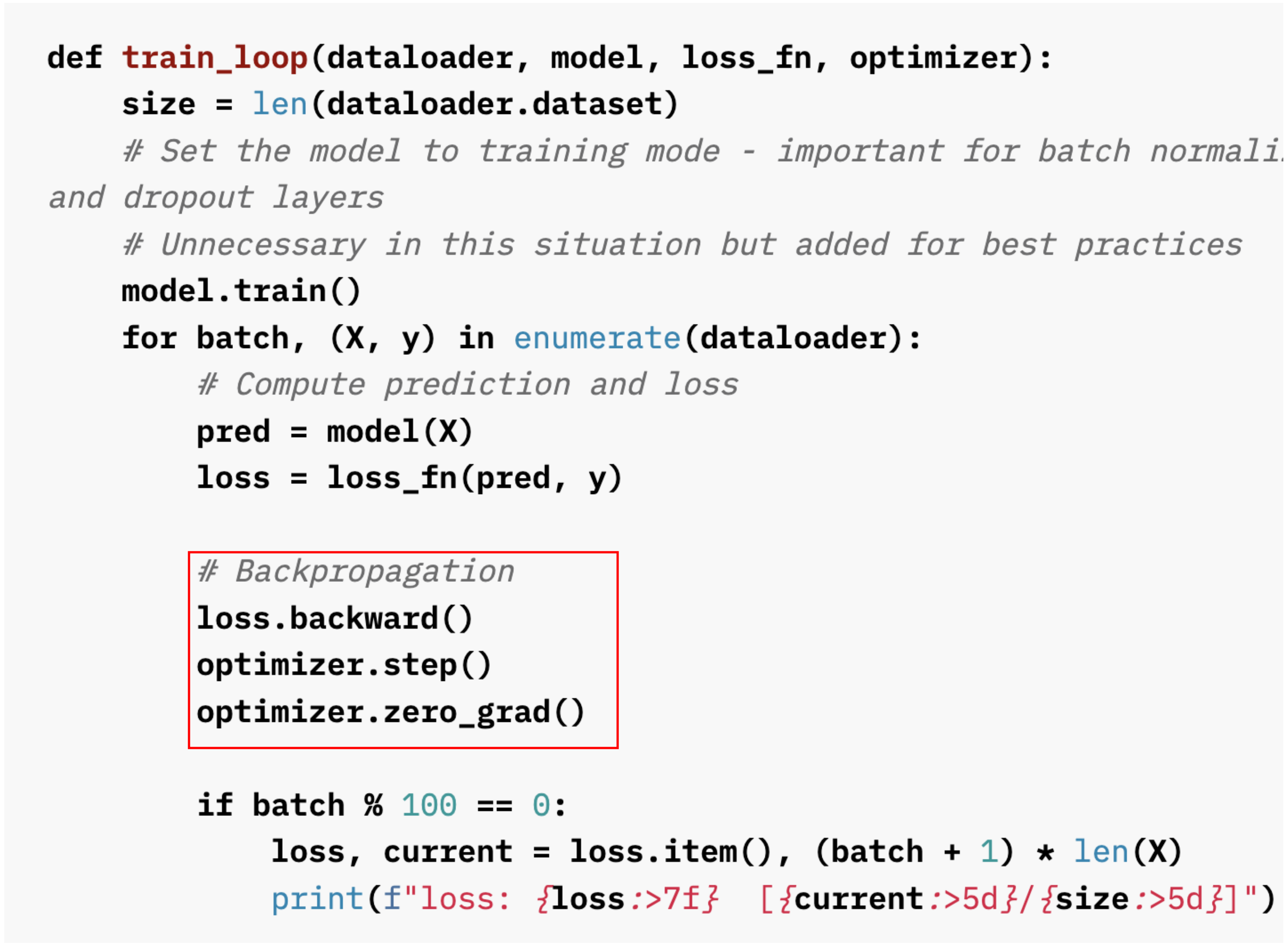
```
For X, Y_gt in TrainLoader:
    Y_pred = Model(X)
    Loss = LossFunction(Y_pred, Y_gt)
    Loss.backward()
    Optimizer.step()
    Optimizer.zero_grad()
```



```python
def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    # Set the model to training mode - important for batch normali
and dropout layers
    # Unnecessary in this situation but added for best practices
    model.train()
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction and loss
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

        if batch % 100 == 0:
            loss, current = loss.item(), (batch + 1) * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
```

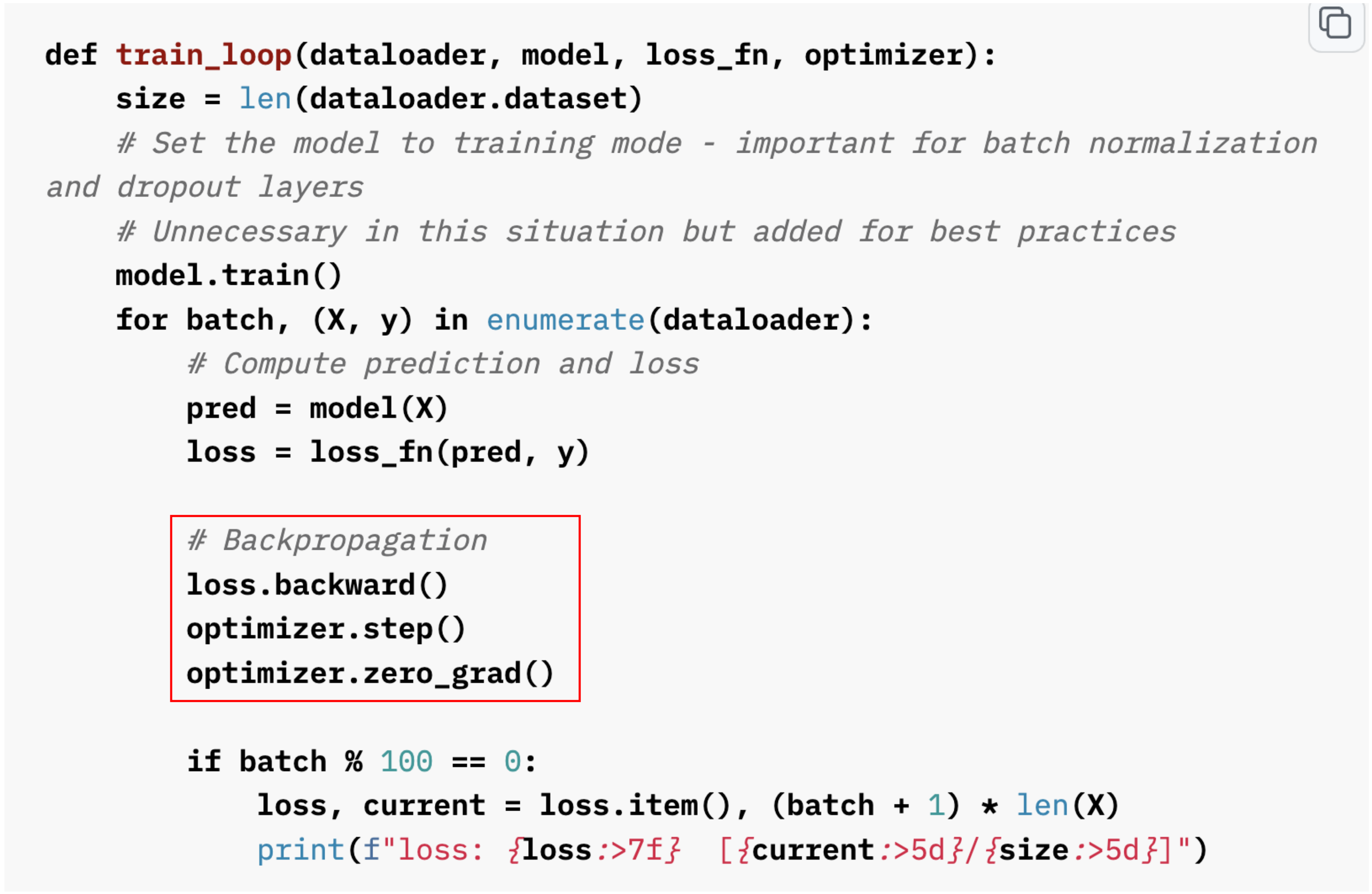
6. Training & Testing

```
For X, Y_gt in TrainLoader:
    Y_pred = Model(X)
    Loss = LossFunction(Y_pred, Y_gt)
    Loss.backward()
    Optimizer.step()
    Optimizer.zero_grad()
```



```python
def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    # Set the model to training mode - important for batch normali
and dropout layers
    # Unnecessary in this situation but added for best practices
    model.train()
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction and loss
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

        if batch % 100 == 0:
            loss, current = loss.item(), (batch + 1) * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
```

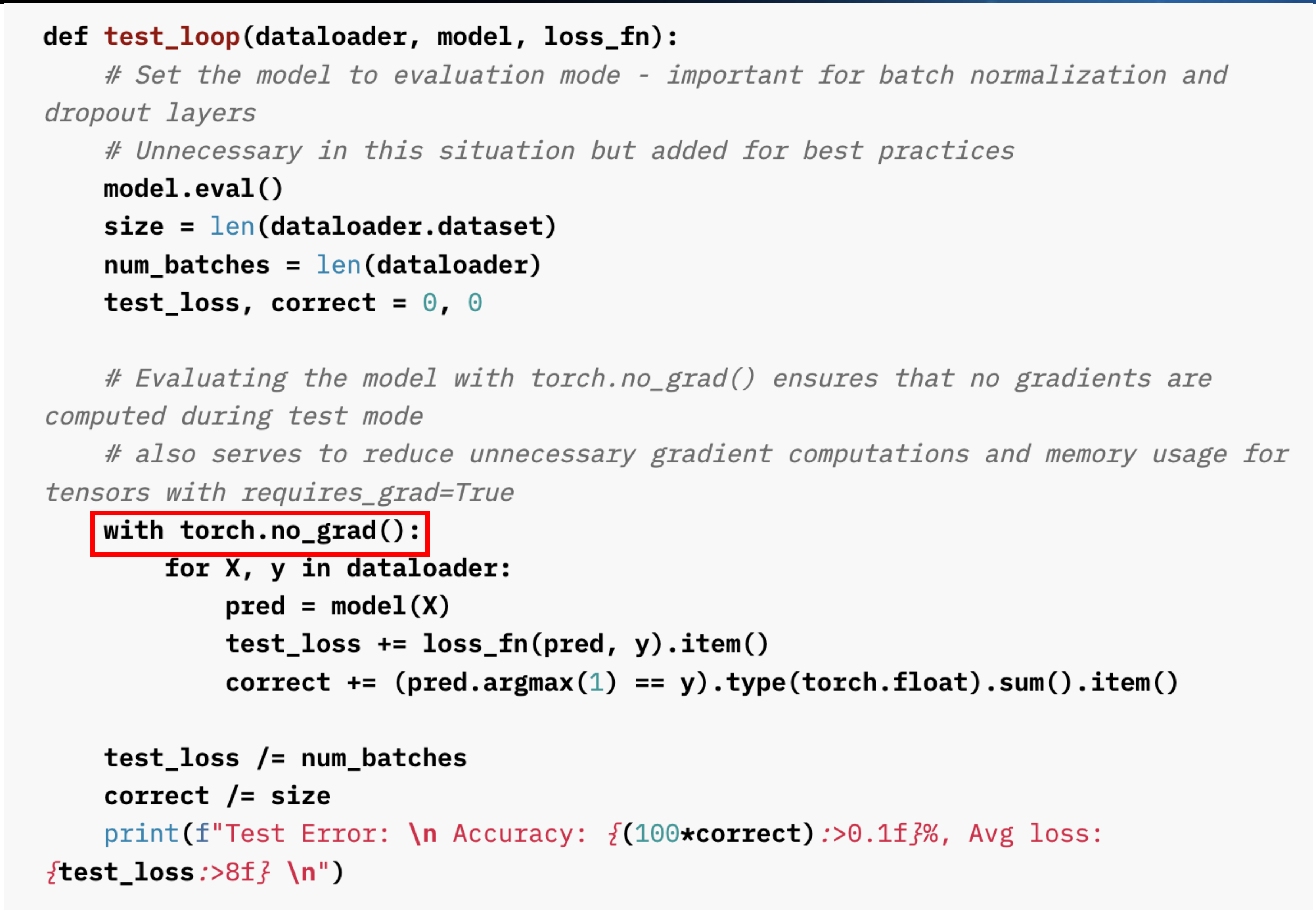
6.1 Training Loop



```
def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    # Set the model to training mode - important for batch normalization
and dropout layers
    # Unnecessary in this situation but added for best practices
    model.train()
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction and loss
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

        if batch % 100 == 0:
            loss, current = loss.item(), (batch + 1) * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
```

[7] Code Credit: https://pytorch.org/tutorials/beginner/basics/optimization_tutorial.html

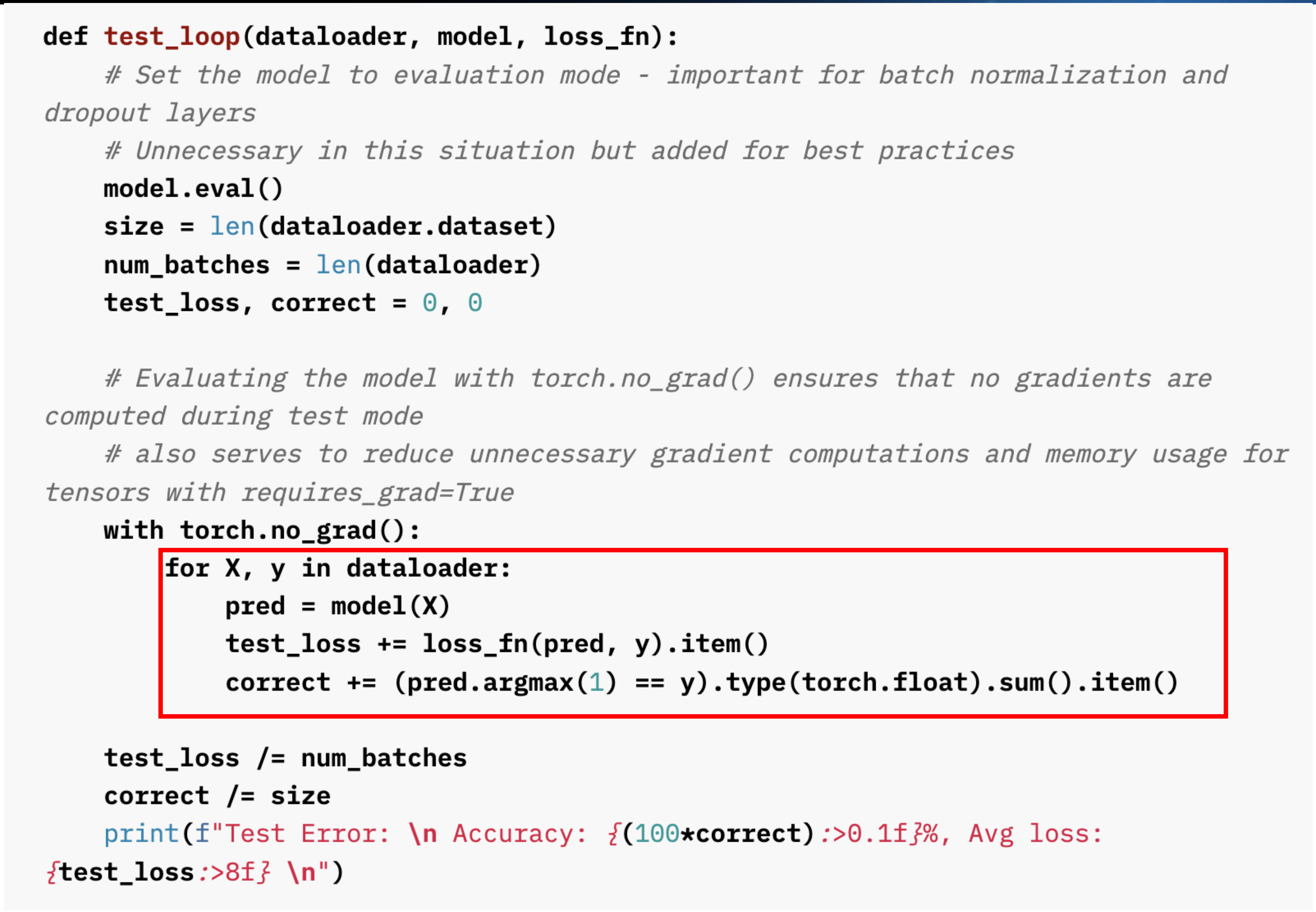
6.2 Test Loop



```
def test_loop(dataloader, model, loss_fn):
    # Set the model to evaluation mode - important for batch normalization and
dropout layers
    # Unnecessary in this situation but added for best practices
    model.eval()
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    test_loss, correct = 0, 0

    # Evaluating the model with torch.no_grad() ensures that no gradients are
computed during test mode
    # also serves to reduce unnecessary gradient computations and memory usage for
tensors with requires_grad=True
    with torch.no_grad():
        for X, y in dataloader:
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()

    test_loss /= num_batches
    correct /= size
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss:
{test_loss:>8f} \n")
```

[7] Code Credit: https://pytorch.org/tutorials/beginner/basics/optimization_tutorial.html

6.2 Test Loop



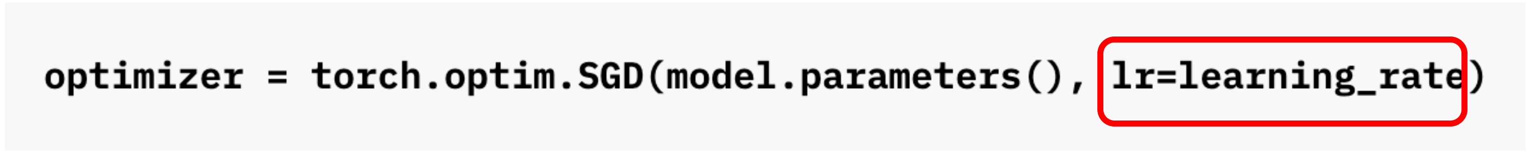
```python
def test_loop(dataloader, model, loss_fn):
    # Set the model to evaluation mode - important for batch normalization and
dropout layers
    # Unnecessary in this situation but added for best practices
    model.eval()
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    test_loss, correct = 0, 0

    # Evaluating the model with torch.no_grad() ensures that no gradients are
computed during test mode
    # also serves to reduce unnecessary gradient computations and memory usage for
tensors with requires_grad=True
    with torch.no_grad():
        for X, y in dataloader:
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()

    test_loss /= num_batches
    correct /= size
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss:
{test_loss:>8f} \n")
```

[7] Code Credit: https://pytorch.org/tutorials/beginner/basics/optimization_tutorial.html

6.3 Hyper Parameters Tuning

Hyperparameters are adjustable parameters that let you control the model optimization process. Different hyperparameter values can impact model training and convergence rates

- Learning Rate

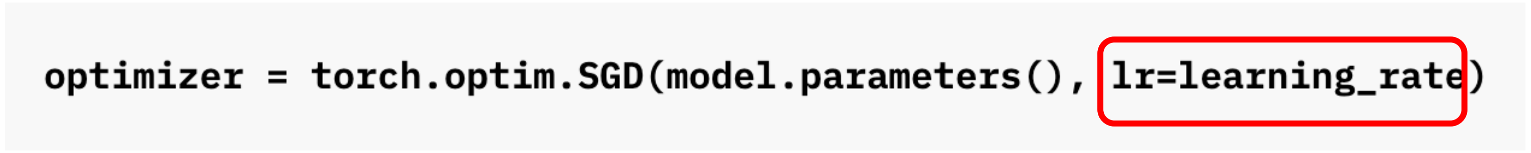


```
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

6.3 Hyper Parameters Tuning

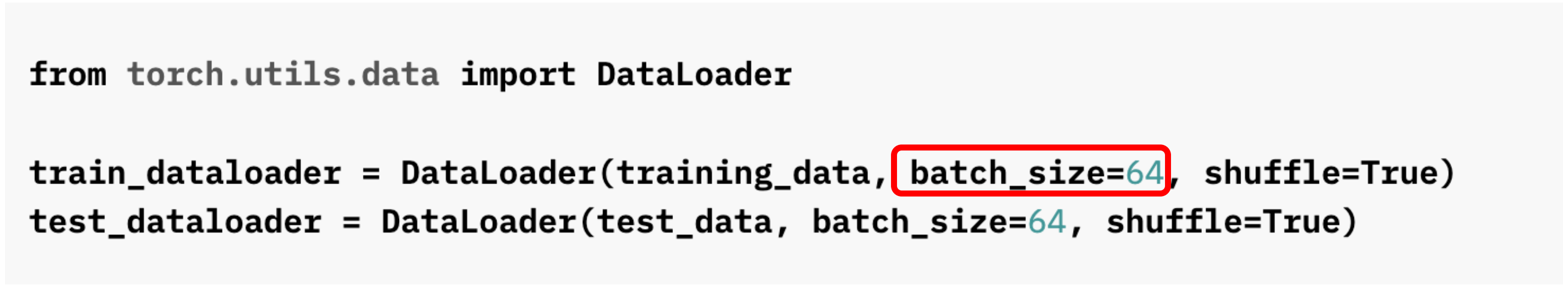
Hyperparameters are adjustable parameters that let you control the model optimization process. Different hyperparameter values can impact model training and convergence rates

- Learning Rate



```
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

- Batch Size

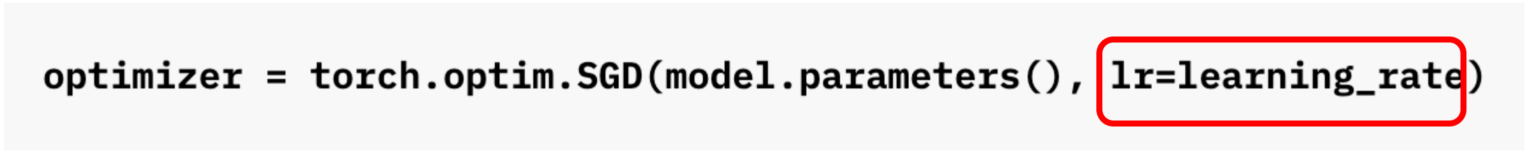


```
from torch.utils.data import DataLoader

train_dataloader = DataLoader(training_data, batch_size=64, shuffle=True)
test_dataloader = DataLoader(test_data, batch_size=64, shuffle=True)
```

6.3 Hyper Parameters Tuning

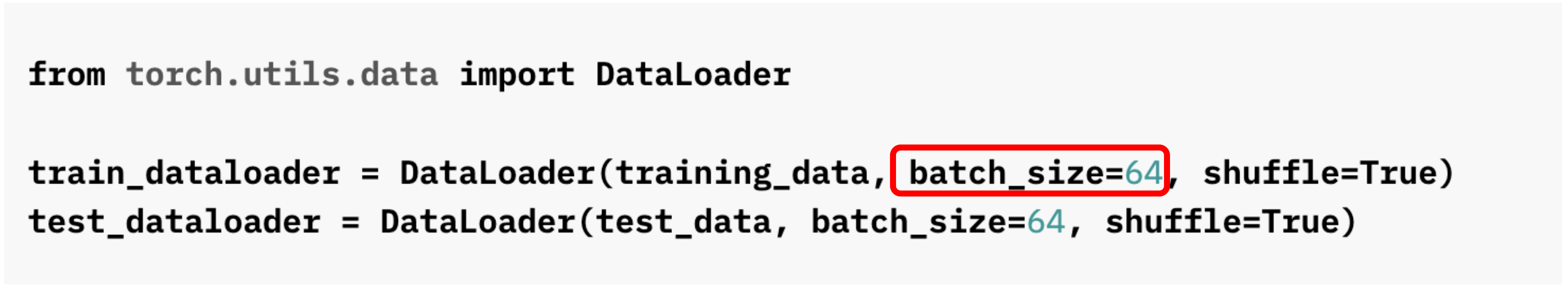
Hyperparameters are adjustable parameters that let you control the model optimization process. Different hyperparameter values can impact model training and convergence rates

- Learning Rate



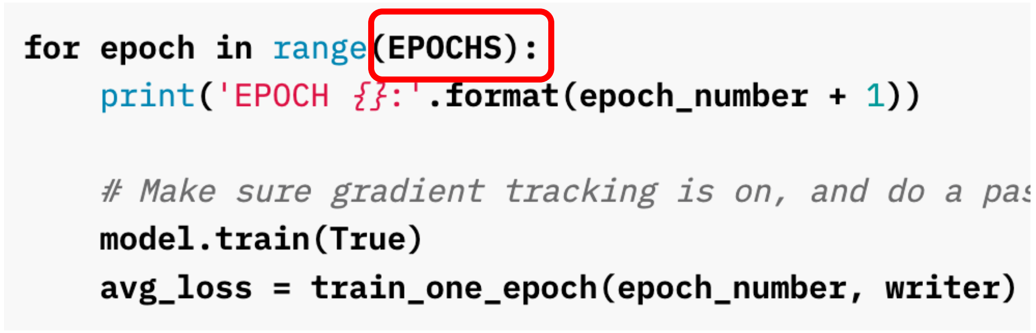
```
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

- Batch Size



```
from torch.utils.data import DataLoader

train_dataloader = DataLoader(training_data, batch_size=64, shuffle=True)
test_dataloader = DataLoader(test_data, batch_size=64, shuffle=True)
```

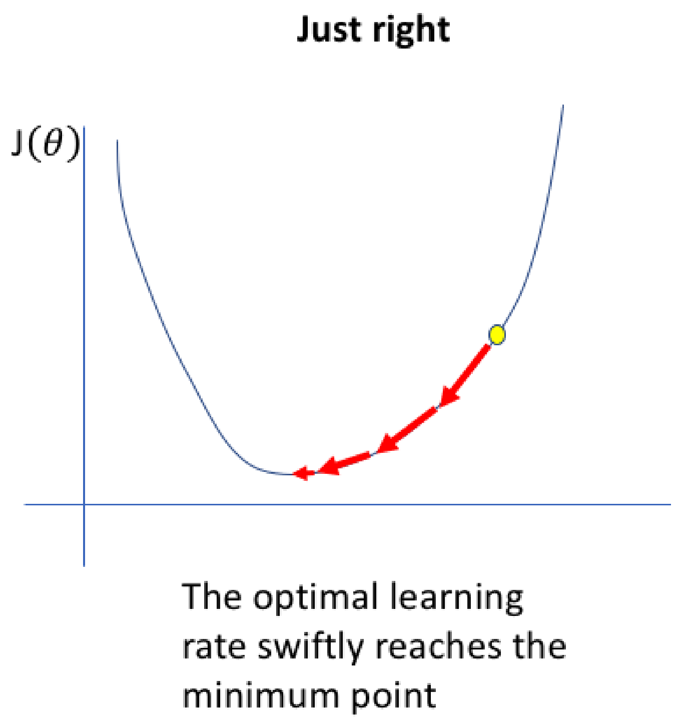
- Number of Epochs



```
for epoch in range(EPOCHS):
    print('EPOCH {}:'.format(epoch_number + 1))

    # Make sure gradient tracking is on, and do a pas
    model.train(True)
    avg_loss = train_one_epoch(epoch_number, writer)
```

6.3.1 Choose your Learning Rate

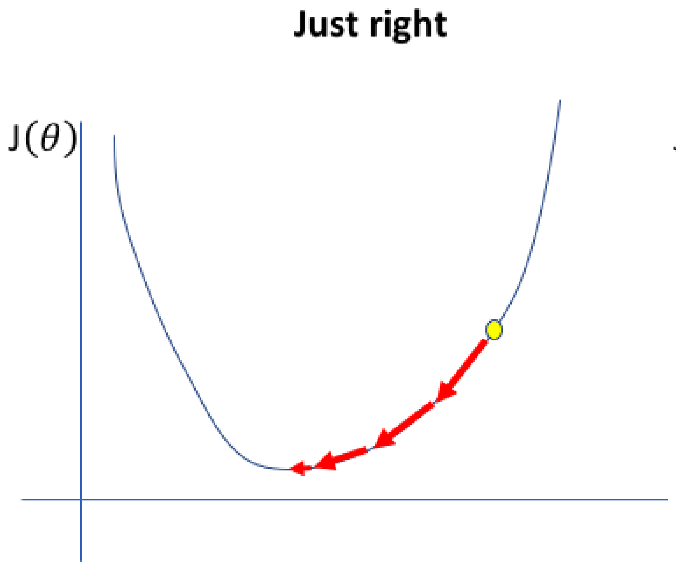
How should we choose learning rate?



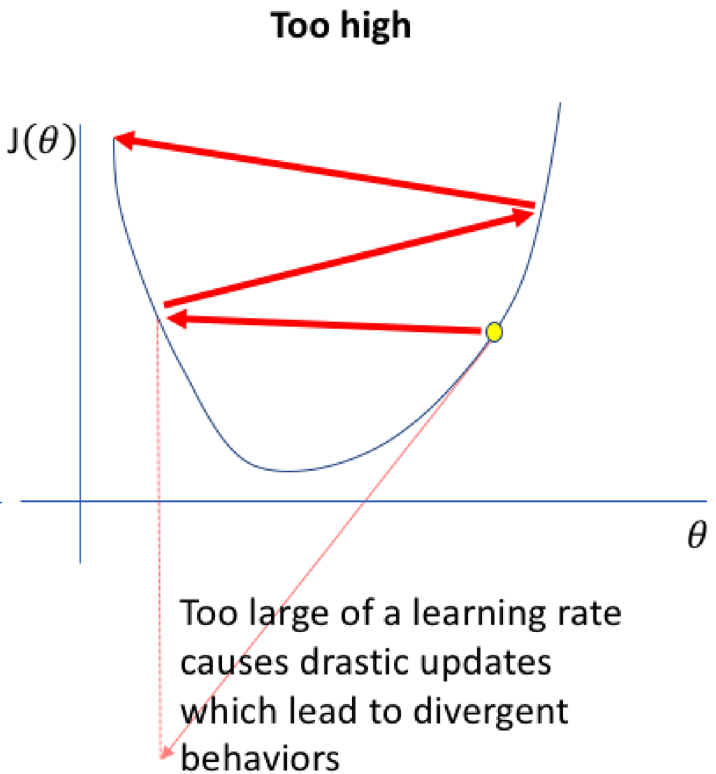
[6] Image Credit: https://medium.com/codex/gradient-descent-cb0f02dc6eab

6.3.1 Choose your Learning Rate

How should we choose learning rate?



The optimal learning rate swiftly reaches the minimum point

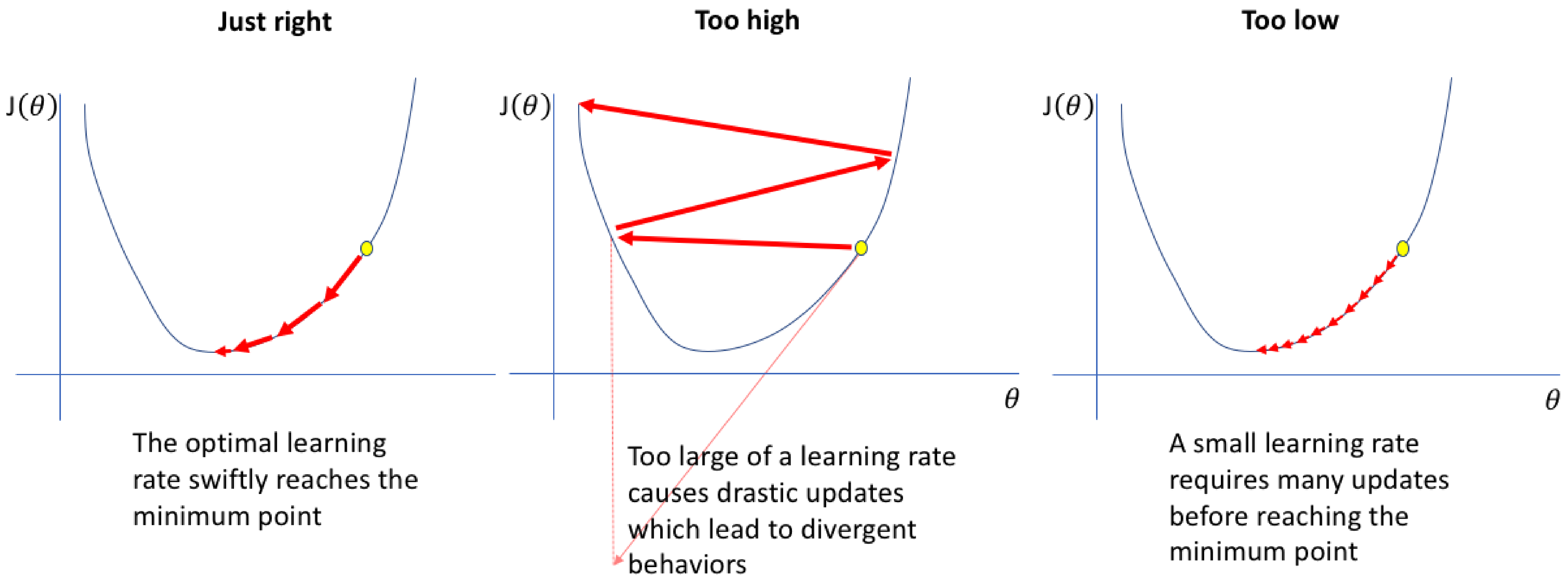


Too large of a learning rate causes drastic updates which lead to divergent behaviors

[6] Image Credit: https://medium.com/codex/gradient-descent-cb0f02dc6eab

6.3.1 Choose your Learning Rate

How should we choose learning rate?



The optimal learning rate swiftly reaches the minimum point

Too large of a learning rate causes drastic updates which lead to divergent behaviors

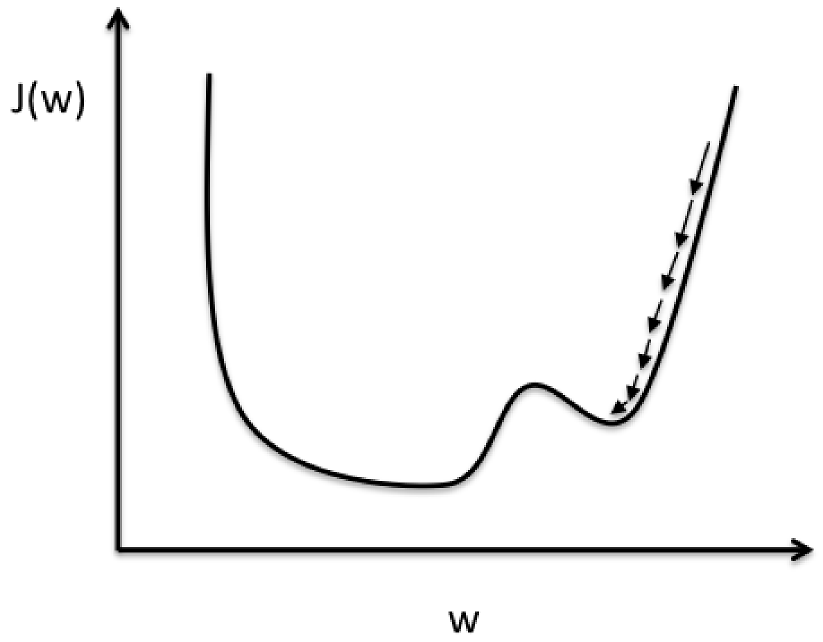
A small learning rate requires many updates before reaching the minimum point

[6] Image Credit: https://medium.com/codex/gradient-descent-cb0f02dc6eab

6.3.1 Choose your Learning Rate

Issue of Learning rate being too small:

For Non-convex Function e.g. Deep Neural Network.

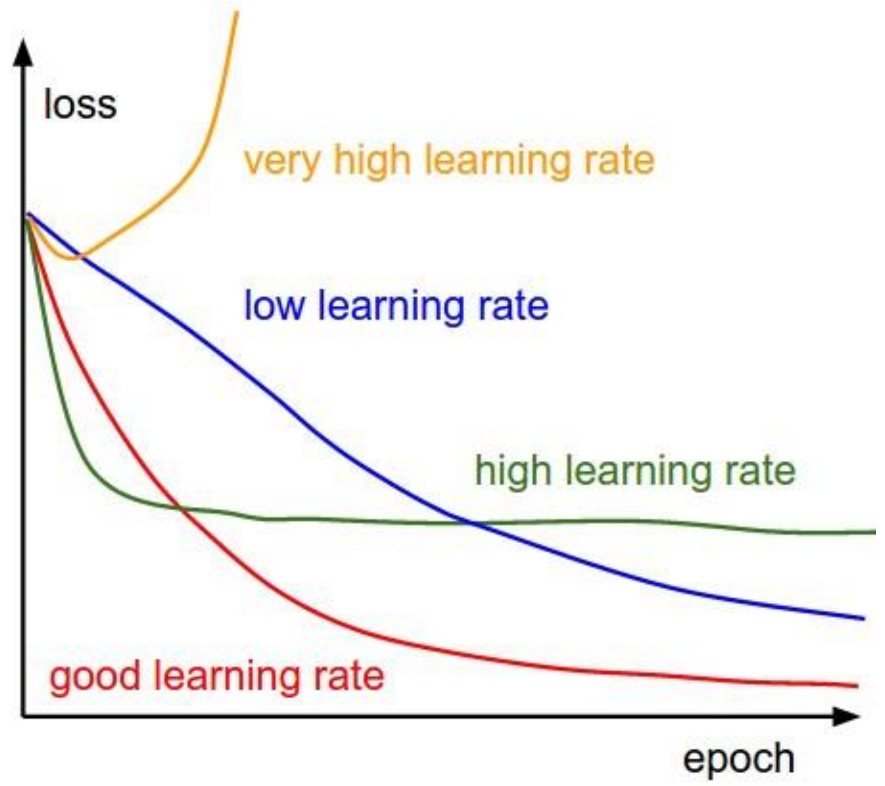


Small learning rate: Many iterations until convergence and trapping in local minima.

[6] Image Credit: https://medium.com/codex/gradient-descent-cb0f02dc6eab

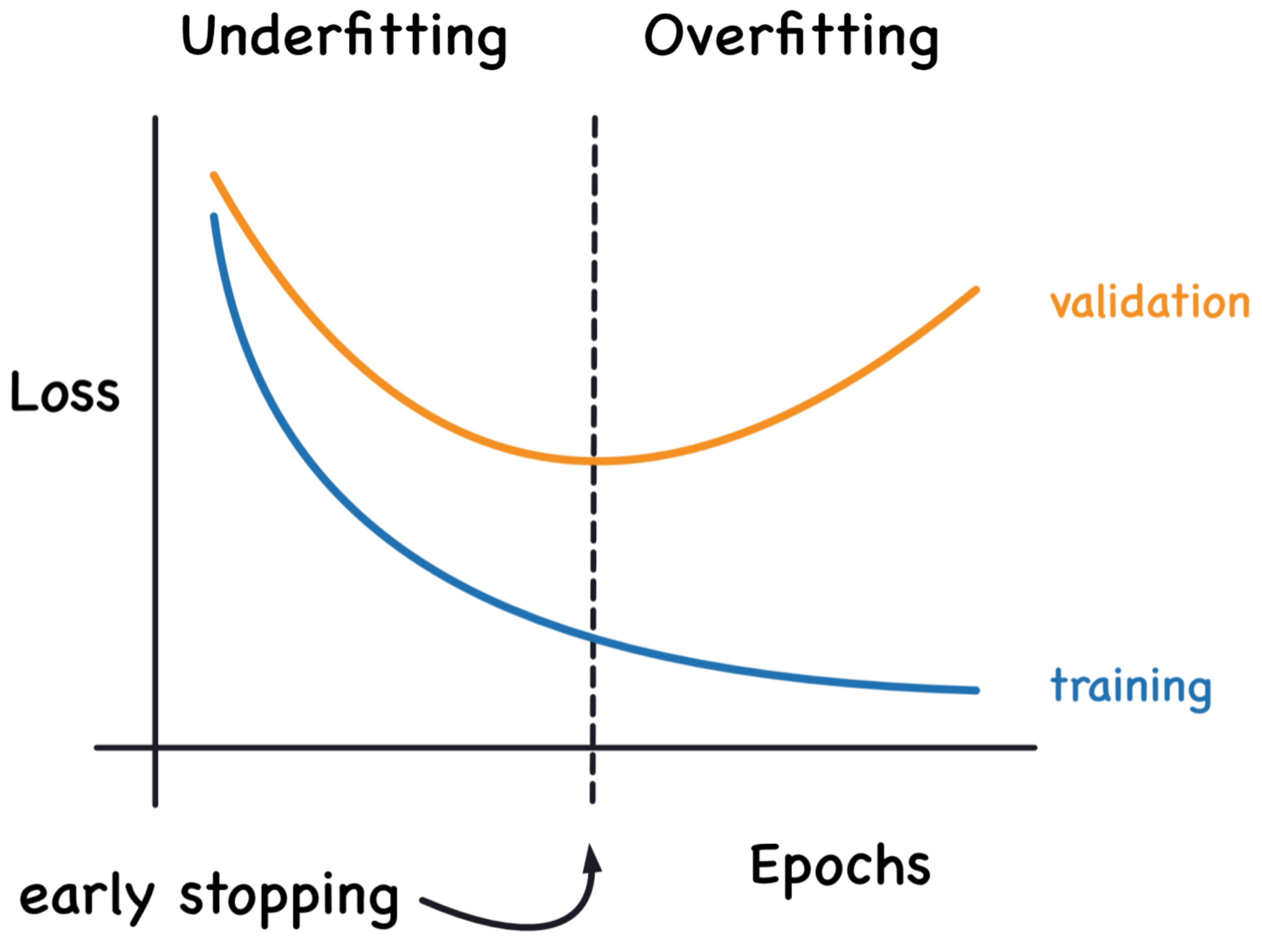
6.3.1 Choose your Learning Rate

Empirical rule to choose your Learning Rate: Observe your Training Loss Curve



[5] Slide Credit: https://www.cs.ubc.ca/~schmidtm/Courses/5XX-S22/S4.pdf

6.3.3 When to stop: Number of Epochs



[8] Image Credit: https://theaisummer.com/regularization/

6.3.3 Adam vs SGD

- SGD is simple and elegant, but is highly **sensitive to learning rate**, and for some data/architectures it **does not converge** :(
- Adam:
  - Basically SGD + **Adaptive Learning Rate** + **Momentum**
  - Much less sensitive to learning rate: Often learning rates of 0.001 or 0.0001 work pretty well.
  - Almost all state-of-the-art models use this.

6.3 Hyper-Parameter Tuning

Empirical Take away:

- Use Batch Size > 1 if you can: reduce the variance of gradient.
- Scale up your Learning Rate as you increase batch size. Use smaller learning rate if you use a small batch size.
- log / Visualize your training Loss and test Loss curve to adjust Learning Rate & Decide when to Early Stop.

6.3 Hyper-Parameter Tuning

Batch Size vs Learning Rate:

- For SGD: If you multiply your batch-size by x you can multiply your learning rate by x
- For Adam: If you multiply your batch-size by x, you can multiply your learning rate by $\sqrt{x}$

Note: these tips are empirical ballparks, and you should still try out different learning rates.

[9] https://www.cs.princeton.edu/~smalladi/blog/2024/01/22/SDEs-ScalingRules/

Topics Not Covered

- Debugging & Analysis Model
  - Profiler
  - Visualizing & Logging: Tensorboard, Wandb
- Distributed Training
  - Use Torch.nn.DistributedDataParallel
  - Optimize your data loading
  - Optimizer / Loss in distributed training
- More Design Detail:
  - Learning rate Scheduler ()
  - Dropout & DropPath & Weight Decay & EMA

Useful Resources

- Recommend reading the tuning playbook by Google:

  https://github.com/google-research/tuning_playbook

- Annotated pytorch implementation for a zoo of models: https://nn.labml.ai/

Thanks for listening and Happy Coding :)

People with no idea about AI saying it will take over the world:
My Neural Network:
Dog

[9] Image Credit: https://www.reddit.com/r/TheInsaneApp/comments/u0a1am/best_machine_learning_meme/

6.3.2 Choose Your Batch Size: A closer look at mini-batch

- We considered stochastic gradient descent (SGD),

$$\boxed{w^{k+1} = w^k - \alpha_k \nabla f_{i_k}(w^k)}.$$

Update Rule of Gradient Descent

which performs a gradient descent step using a random training example i_k.

  - This gives an unbiased gradient approximation, $\mathbb{E}[\nabla f_{i_k}(w^k)] = \nabla f(w^k)$.

[5] Slide Credit: https://www.cs.ubc.ca/~schmidtm/Courses/5XX-S22/S4.pdf

6.3.2 Choose Your Batch Size

- We considered stochastic gradient descent (SGD),

$$w^{k+1} = w^k - \alpha_k \nabla f_{i_k}(w^k).$$

which performs a gradient descent step using a random training example i_k.
  - This gives an unbiased gradient approximation, $\mathbb{E}[\nabla f_{i_k}(w^k)] = \nabla f(w^k)$.

- Deterministic gradient descent uses all n gradients,

$$\boxed{\nabla f(w^k) = \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(w^k).}$$

Gradient Estimation of ALL n Samples

[5] Slide Credit: https://www.cs.ubc.ca/~schmidtm/Courses/5XX-S22/S4.pdf

6.3.2 Choose Your Batch Size

- We considered stochastic gradient descent (SGD),

$$w^{k+1} = w^k - \alpha_k \nabla f_{i_k}(w^k).$$

which performs a gradient descent step using a random training example i_k.
  - This gives an unbiased gradient approximation, $\mathbb{E}[\nabla f_{i_k}(w^k)] = \nabla f(w^k)$.
- Deterministic gradient descent uses all n gradients,

$$\nabla f(w^k) = \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(w^k).$$

- A common variant is to use m samples as a mini-batch $\mathcal{B}^k$,

$$\boxed{\nabla f(w^k) \approx \frac{1}{m}\sum_{i \in \mathcal{B}^k} \nabla f_i(w^k)}$$

Approximate to the True gradient with m samples in Current Batch.

This approximate is unbiased given samples are IID

[5] Slide Credit: https://www.cs.ubc.ca/~schmidtm/Courses/5XX-S22/S4.pdf

6.3.2 Choose Your Batch Size

- With m samples in our mini-batch we have that (see bonus)

$$\boxed{\mathbb{E}[\|e^k\|^2] = \frac{\sigma(w^k)^2}{m}}$$

The variance of gradient with batch size M

where $\sigma^2(w^k)$ is the variation in the individual gradients at w^k.

[5] Slide Credit: https://www.cs.ubc.ca/~schmidtm/Courses/5XX-S22/S4.pdf ; See proof from slide 21.

6.3.2 Choose Your Batch Size

- With m samples in our mini-batch we have that (see bonus)

$$\mathbb{E}[\|e^k\|^2] = \frac{\sigma(w^k)^2}{m},$$

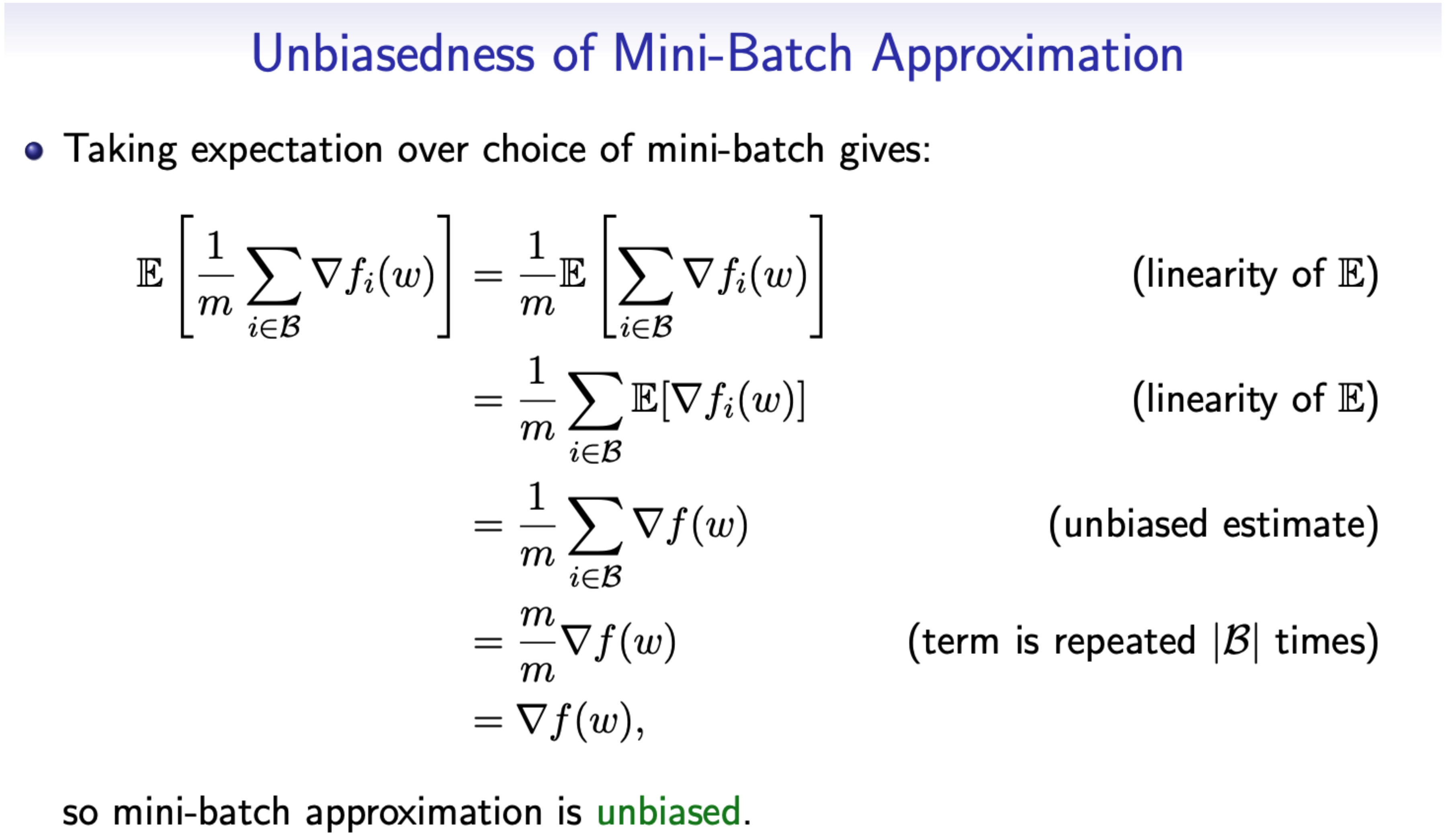
where $\sigma^2(w^k)$ is the variation in the individual gradients at w^k.

- "With a mini-batch size of 100, effect of noise is divided by 100".
  - Biggest gains obtained for increasing small batch sizes.
- "With a mini-batch size of 100, you can use a step size that is 100-times larger."
  - "Linear scaling rule" (but may not guarantee progress if $\alpha_k \geq 2/L$)

$$w^{k+1} = w^k - \alpha_k \nabla f_{i_k}(w^k).$$

[5] Slide Credit: https://www.cs.ubc.ca/~schmidtm/Courses/5XX-S22/S4.pdf

Prove of the Unbiasedness and variance for mini-batch SGD



Unbiasedness of Mini-Batch Approximation

- Taking expectation over choice of mini-batch gives:

$$
\begin{aligned}
\mathbb{E}\left[\frac{1}{m}\sum_{i\in\mathcal{B}}\nabla f_i(w)\right] &= \frac{1}{m}\mathbb{E}\left[\sum_{i\in\mathcal{B}}\nabla f_i(w)\right] && \text{(linearity of } \mathbb{E}\text{)}\\
&= \frac{1}{m}\sum_{i\in\mathcal{B}}\mathbb{E}[\nabla f_i(w)] && \text{(linearity of } \mathbb{E}\text{)}\\
&= \frac{1}{m}\sum_{i\in\mathcal{B}}\nabla f(w) && \text{(unbiased estimate)}\\
&= \frac{m}{m}\nabla f(w) && \text{(term is repeated } |\mathcal{B}| \text{ times)}\\
&= \nabla f(w),
\end{aligned}
$$

so mini-batch approximation is unbiased.

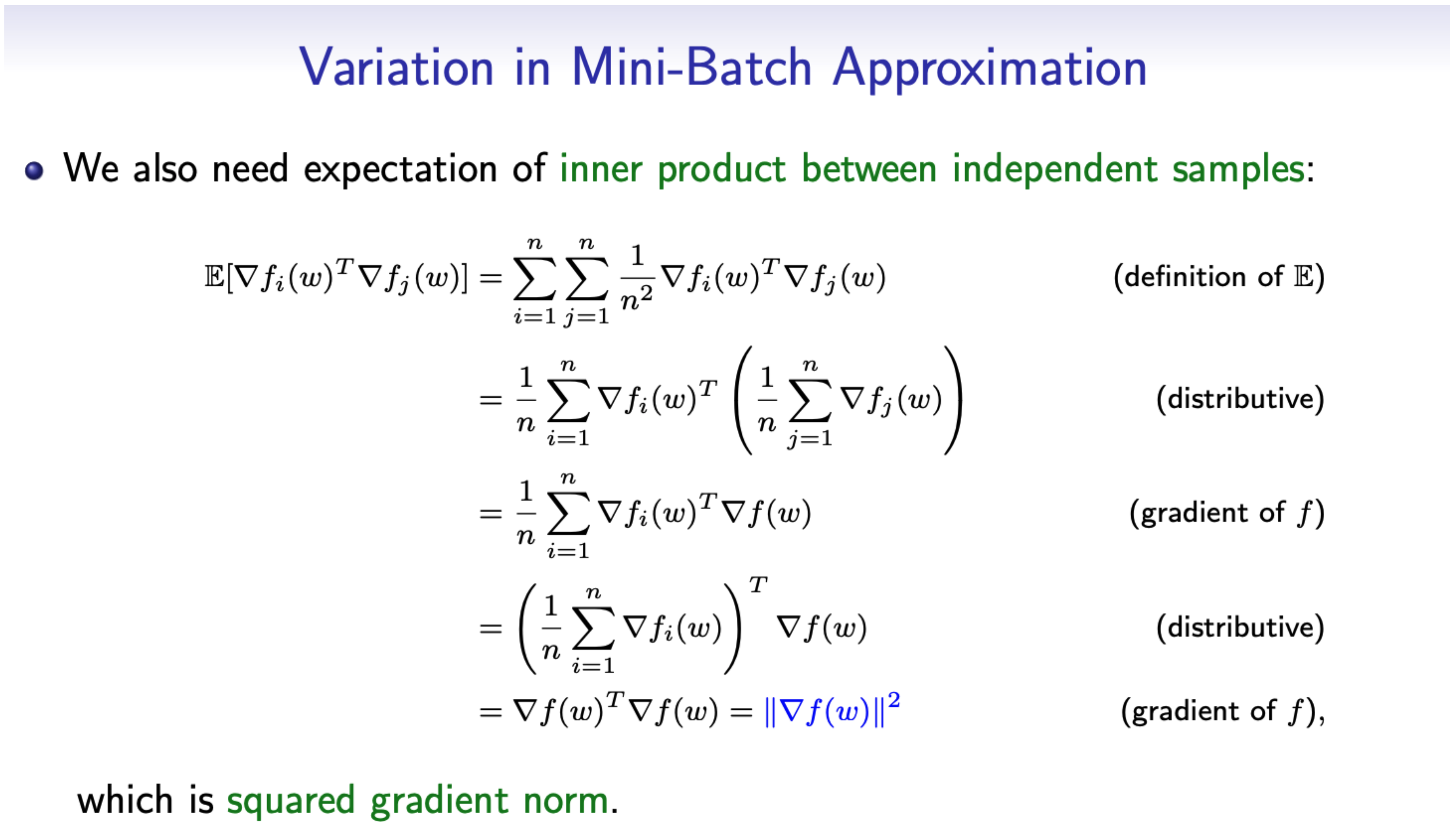
[5] Slide Credit: https://www.cs.ubc.ca/~schmidtm/Courses/5XX-S22/S4.pdf

Variation in Mini-Batch Approximation

- To analyze variation in gradients, we use a variance-like identity:
  - If random variable g is an unbiased approximation of vector μ, then

$$
\begin{aligned}
\mathbb{E}[\|g-\mu\|^2] &= \mathbb{E}[\|g\|^2 - 2g^T\mu + \|\mu\|^2] && \text{(expand square)}\\
&= \mathbb{E}[\|g\|^2] - 2\mathbb{E}[g]^T\mu + \|\mu\|^2 && \text{(linearity of } \mathbb{E}\text{)}\\
&= \mathbb{E}[\|g\|^2] - 2\mu^T\mu + \|\mu\|^2 && \text{(unbiased)}\\
&= \mathbb{E}[\|g\|^2] - \|\mu\|^2.
\end{aligned}
$$

[5] Slide Credit: https://www.cs.ubc.ca/~schmidtm/Courses/5XX-S22/S4.pdf



Variation in Mini-Batch Approximation

- We also need expectation of inner product between independent samples:

$$
\begin{aligned}
\mathbb{E}[\nabla f_i(w)^T \nabla f_j(w)] &= \sum_{i=1}^{n}\sum_{j=1}^{n} \frac{1}{n^2} \nabla f_i(w)^T \nabla f_j(w) && \text{(definition of } \mathbb{E}) \\
&= \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(w)^T \left(\frac{1}{n}\sum_{j=1}^{n} \nabla f_j(w)\right) && \text{(distributive)} \\
&= \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(w)^T \nabla f(w) && \text{(gradient of } f) \\
&= \left(\frac{1}{n}\sum_{i=1}^{n} \nabla f_i(w)\right)^T \nabla f(w) && \text{(distributive)} \\
&= \nabla f(w)^T \nabla f(w) = \|\nabla f(w)\|^2 && \text{(gradient of } f),
\end{aligned}
$$

which is squared gradient norm.

[5] Slide Credit: https://www.cs.ubc.ca/~schmidtm/Courses/5XX-S22/S4.pdf

Variation Bound for Mini-Batch Approximation

- Let $g_2(w) = \frac{1}{2}(\nabla f_i(w) + \nabla f_j(w))$ be mini-batch approximation with 2 samples.

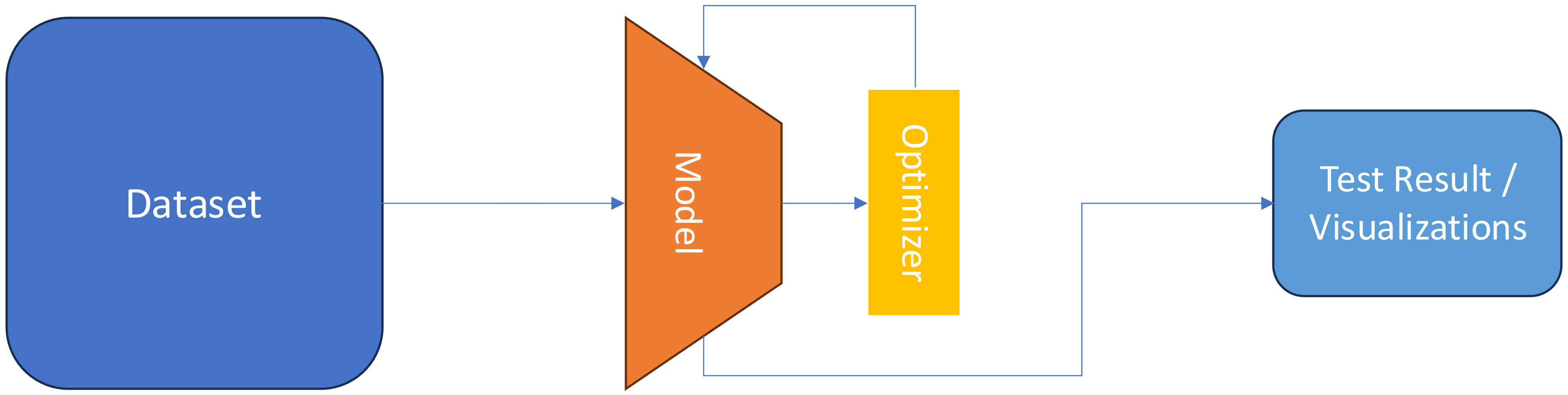
$$\begin{aligned}
\mathbb{E}[\|g_2(w) - \nabla f(w)\|^2] &= \mathbb{E}[\|\frac{1}{2}(\nabla f_i(w) + \nabla f_j(w))\|^2] - \|\nabla f(w)\|^2 && \text{(variance identity)} \\
&= \frac{1}{4}\mathbb{E}[\|\nabla f_i(w)\|^2] + \frac{1}{2}\mathbb{E}[\nabla f_i(w)^T \nabla f_j(w)] + \frac{1}{4}\mathbb{E}[\|\nabla f_j(w)\|^2] - \|\nabla f(w)\|^2 && \text{(expand square)} \\
&= \frac{1}{2}\mathbb{E}[\|\nabla f_i(w)\|^2] + \frac{1}{2}\mathbb{E}[\nabla f_i(w)^T \nabla f_j(w)] - \|\nabla f(w)\|^2 && (\mathbb{E}[\nabla f_i] = \mathbb{E}[\nabla f_j]) \\
&= \frac{1}{2}\mathbb{E}[\|\nabla f_i(w)\|^2] + \frac{1}{2}\|\nabla f(w)\|^2 - \|\nabla f(w)\|^2 && (\mathbb{E}[\nabla f_i \nabla f_j] = \nabla f^2) \\
&= \frac{1}{2}\mathbb{E}[\|\nabla f_i(w)\|^2] - \frac{1}{2}\|\nabla f(w)\|^2 \\
&= \frac{1}{2}\left(\mathbb{E}[\|\nabla f_i(w)\|^2] - \|\nabla f(w)\|^2\right) && \text{(factor } \tfrac{1}{2}\text{)} \\
&= \frac{1}{2}\mathbb{E}[\|\nabla f_i(w) - \nabla f(w)\|^2] && \text{(variance identity)} \\
&= \frac{\sigma(w)^2}{2} && (\sigma^2 \text{ is 1-sample variation})
\end{aligned}$$

- So SGD error $\mathbb{E}[\|e^k\|^2]$ is cut in half compared to using 1 sample.

[5] Slide Credit: https://www.cs.ubc.ca/~schmidtm/Courses/5XX-S22/S4.pdf

Overview

- Fundamentals Pipelines of a DL model



1. Preparing Your Data

2. Train the model

3. Test the model