Introduction to Deep Learning

Tutorial 4

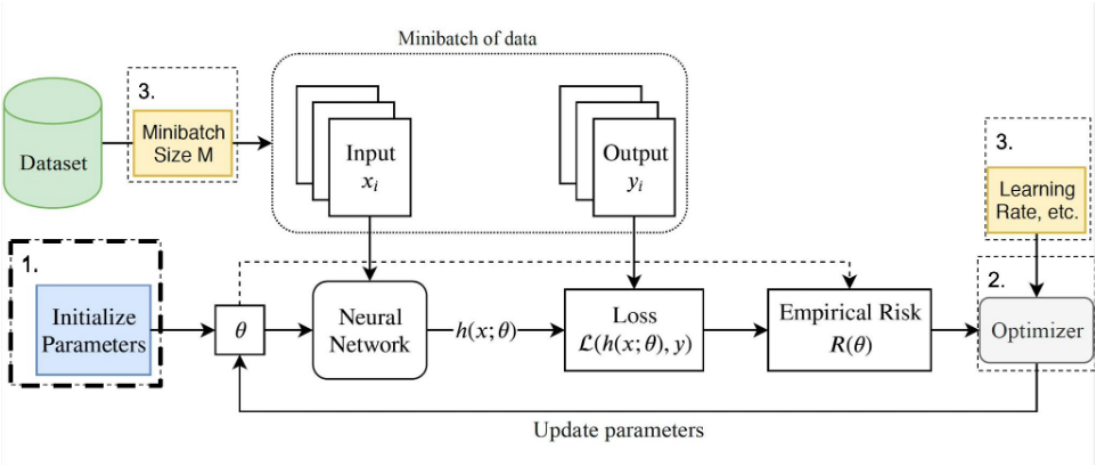
Gabriel Deza
Department of Industrial Engineering
Tel Aviv University

November 24, 2025

Overview

- Overall training loop
    - Initialization
    - Optimization:
        - Gradient Descent
        - Momentum, Nesterov accelerated Momentum
        - Learning rate schedulers: Adagrad, RMSProp, Adam
    - Hyperparameter tuning

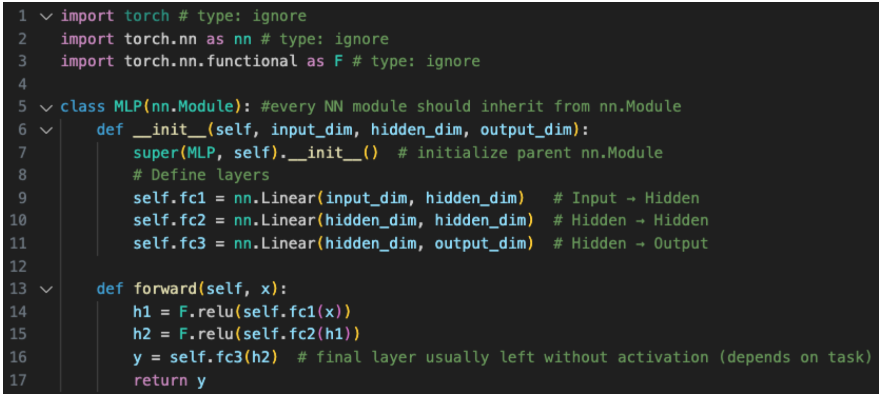
Neural Network Training Loop



Initializing weights

TLDR: The art of weight initialization is already largely solved.

Let PyTorch handle it unless you know what you are doing.



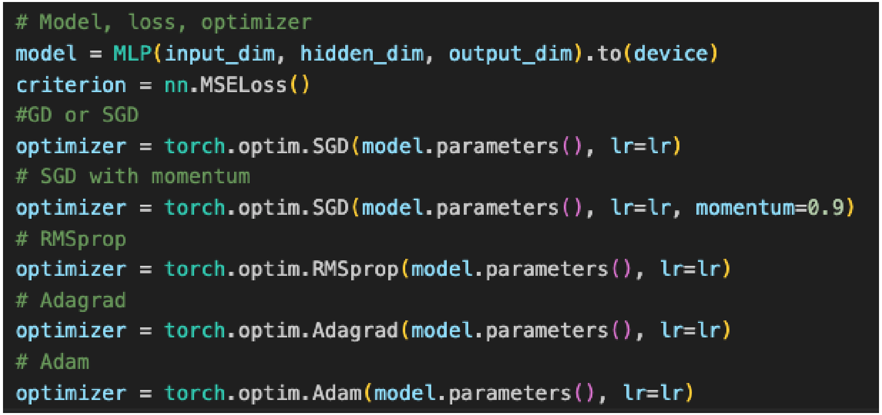
```python
import torch # type: ignore
import torch.nn as nn # type: ignore
import torch.nn.functional as F # type: ignore

class MLP(nn.Module): #every NN module should inherit from nn.Module
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(MLP, self).__init__()  # initialize parent nn.Module
        # Define layers
        self.fc1 = nn.Linear(input_dim, hidden_dim)   # Input → Hidden
        self.fc2 = nn.Linear(hidden_dim, hidden_dim)  # Hidden → Hidden
        self.fc3 = nn.Linear(hidden_dim, output_dim)  # Hidden → Output

    def forward(self, x):
        h1 = F.relu(self.fc1(x))
        h2 = F.relu(self.fc2(h1))
        y = self.fc3(h2)  # final layer usually left without activation (depends on task)
        return y
```

Optimization at 3000 feet

- TLDR: use Adam.
- If you have time, try different optimizers/hyperparameters.



```python
# Model, loss, optimizer
model = MLP(input_dim, hidden_dim, output_dim).to(device)
criterion = nn.MSELoss()
#GD or SGD
optimizer = torch.optim.SGD(model.parameters(), lr=lr)
# SGD with momentum
optimizer = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9)
# RMSprop
optimizer = torch.optim.RMSprop(model.parameters(), lr=lr)
# Adagrad
optimizer = torch.optim.Adagrad(model.parameters(), lr=lr)
# Adam
optimizer = torch.optim.Adam(model.parameters(), lr=lr)
```

Optimization formally

- Given a training set: $\{(x_1, y_1), \ldots, (x_n, y_n)\}$
- Prediction function: $h(x; \theta)$
- Define a loss function: $\mathcal{L}(h(x; \theta), y)$
- Find the parameters: $\theta = (\theta_1, \ldots, \theta_k)$ which minimizes the **empirical risk** $R(\theta)$:

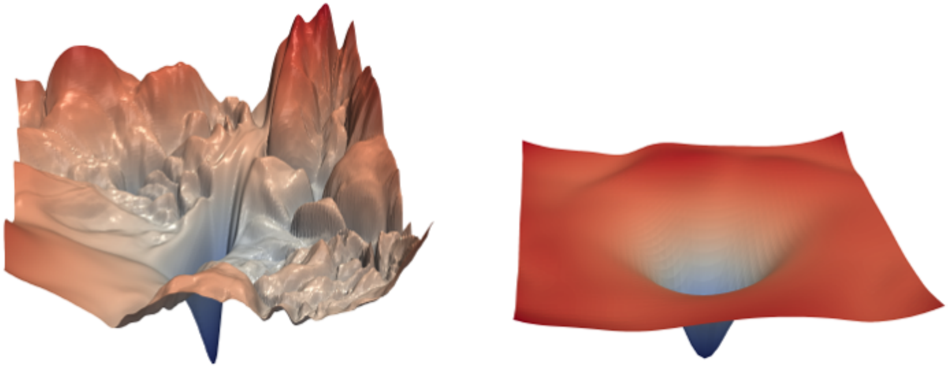
$$\min_\theta R(\theta) \; = \; \min_\theta \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}\big(h(x_i; \theta), \; y_i\big)$$

Optimum satisfies $\nabla R(\theta^*) = 0$

Skip Connections (Residuals)

Skip Connections: $x_{l+1} = F_l(x_l) \to x_{l+1} = x_l + F_l(x_l)$

- Learn a residual correction instead of a full mapping.
- Identity path helps gradients flow $\Rightarrow$ easier deep training.



(a) without skip connections (b) with skip connections

Gradient Descent

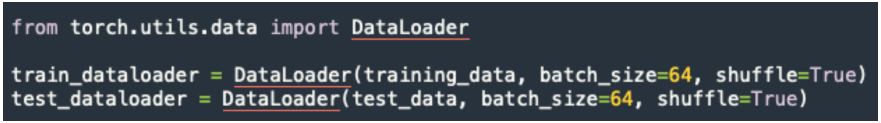
Geometric view:

- Gradient is orthogonal to the level set.
- The negative gradient gives the steepest descent direction.

Update rule:

$$\theta^{(k+1)} = \theta^{(k)} - \eta \nabla R(\theta^{(k)})$$

SGD: Uses random mini-batches to approximate the gradient. (GD is SGD with the full dataset as one batch.)



```python
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)
```

Gradient Descent with Momentum

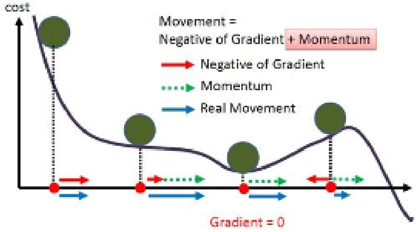
- Initialize parameters $\theta^{(0)}$ and momentum $\delta^{(0)} = \mathbf{0}$.
- For $k = 0, 1, 2, \ldots$ until convergence:
    - **Update momentum:**

$$\delta^{(k+1)} = -\eta \nabla R(\theta^{(k)}) + \alpha \delta^{(k)}$$

    - **Update parameters:**

$$\theta^{(k+1)} = \theta^{(k)} + \delta^{(k+1)}$$

- **Pros:** accelerates learning by accumulating a velocity from past gradients; helps damp oscillations. Use $\alpha = 0.9$.



AdaGrad Optimizer

- Initialize parameters $\theta^{(0)}$ randomly and initialize accumulator $G^0 = 0$
- For each iteration $k = 0, 1, 2, \dots$:
  - Accumulate squared gradients:

$$G_i^{(k+1)} = G_i^{(k)} + \nabla R(\theta_i^{(k)})^2$$

  - Update parameters:

$$\theta_i^{(k+1)} = \theta_i^{(k)} - \frac{\eta}{\sqrt{G_i^{(k)}} + \epsilon} \odot \nabla R(\theta_i^{(k)})$$

- G is a vector of how much each parameter has changed over time.
- **Intuition:** increases the learning rate for sparse features and decreases it for frequent features, based on gradient history.

RMSProp/Adadelta Optimizer

- Maintain an exponential moving average of squared gradients:

$$G_i^{(k+1)} = \gamma G_i^{(k)} + (1-\gamma)\nabla R(\theta_i^{(k)})^2$$

- Parameter update:

$$\theta_i^{(k+1)} = \theta_i^{(k)} - \frac{\eta}{\sqrt{G_i^{(k+1)}} + \epsilon} \nabla R(\theta_i^{(k)})$$

- **Intuition:** Like AdaGrad, adapts learning rates per parameter, but emphasizes recent gradients ($\gamma \approx 0.9$).
- **Pros:** Prevents learning rates from vanishing; stabilizes training.

Adam Optimizer

- Maintain exponential moving averages of gradients and squared gradients:

$$m_k = \beta_1 m_{k-1} + (1-\beta_1)\nabla R(\theta^{(k)}), \quad v_k = \beta_2 v_{k-1} + (1-\beta_2)(\nabla R(\theta^{(k)}))^2$$

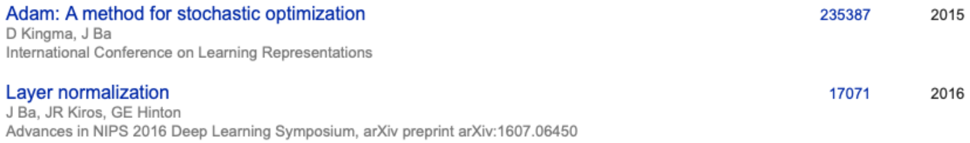
- Apply bias correction:

$$\hat{m}_k = \frac{m_k}{1-\beta_1^k}, \quad \hat{v}_k = \frac{v_k}{1-\beta_2^k}$$

- Parameter update:

$$\theta_{k+1} = \theta_k - \frac{\eta}{\sqrt{\hat{v}_k} + \epsilon}\hat{m}_k$$

- **Intuition:** Combines Momentum (via m_k) and RMSProp (via v_k). Adaptive learning rates + bias correction = fast and stable training.
- **Defaults:** $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$.



Adam: A method for stochastic optimization 235387 2015
D Kingma, J Ba
International Conference on Learning Representations

Layer normalization 17071 2016
J Ba, JR Kiros, GE Hinton
Advances in NIPS 2016 Deep Learning Symposium, arXiv preprint arXiv:1607.06450

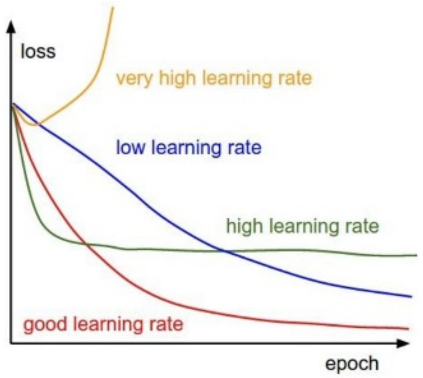
Learning Rate

Ideal Learning Rate should be:

- Should not be too big (objective will blow up)
- Should not be too small (takes longer to converge)

Convergence criteria:

- Change in objective function is close to zero
- Gradient norm is close to zero
- Validation error starts to increase (early-stopping)



Idealized cartoon depiction of different learning rates.
Image credit: Andrej Karpathy

Learning Rate Scheduling: Brief Overview

Anneal (decay) learning rate over time so the parameters can settle into a local minimum.
Typical decay strategies:

1. **Decay:** reduce the learning rate somehow monotonically over time (many possible ways).
2. **Cosine Scheduling:** decay slowly, then quickly, then slowly again. Not clear why this works well.
3. **Schedule-Free Tuning:** uses clever math to tune the learning rate automatically.

Batch Size

Definition: Number of training examples used to compute the gradient at each iteration.

- Typical small batch sizes are powers of 2: 32, 64, 128, 256, 512
- Large batches are in the thousands

TLDR: Pick largest batch size your GPU/TPU setup supports

Large Batch Size effects:

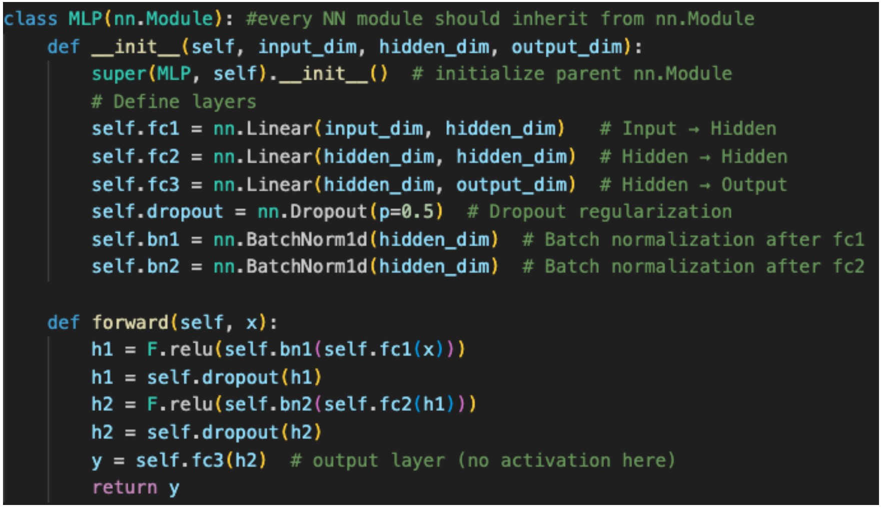
- Fewer parameter updates per epoch
- More accurate gradient estimate
- Better parallelization efficiency $\Rightarrow$ faster wallclock training
- **May hurt generalization** (risk of sharper minima or poorer local optima)

Architecture Choice

- **Top-level rule:** Start with a simple, proven architecture. Add complexity only if needed.
- Avoid chasing every “new thing” unless it clearly helps.
- With experience, you’ll develop intuition about which techniques work best for particular tasks.

Common Bells and Whistles:

- Layer Normalization
- Batch Normalization
- Dropout
- Residual Connections
- Extensive hyperparameter tuning



```
class MLP(nn.Module): #every NN module should inherit from nn.Module
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(MLP, self).__init__()  # initialize parent nn.Module
        # Define layers
        self.fc1 = nn.Linear(input_dim, hidden_dim)   # Input → Hidden
        self.fc2 = nn.Linear(hidden_dim, hidden_dim)  # Hidden → Hidden
        self.fc3 = nn.Linear(hidden_dim, output_dim)  # Hidden → Output
        self.dropout = nn.Dropout(p=0.5)  # Dropout regularization
        self.bn1 = nn.BatchNorm1d(hidden_dim)  # Batch normalization after fc1
        self.bn2 = nn.BatchNorm1d(hidden_dim)  # Batch normalization after fc2

    def forward(self, x):
        h1 = F.relu(self.bn1(self.fc1(x)))
        h1 = self.dropout(h1)
        h2 = F.relu(self.bn2(self.fc2(h1)))
        h2 = self.dropout(h2)
        y = self.fc3(h2)  # output layer (no activation here)
        return y
```

Hyperparameter Optimization (HPO)

- Many settings (LR, batch size, weight decay, depth, dropout, scheduler) strongly affect performance.
- HPO frameworks (e.g., **Optuna**, Ray Tune, Hyperopt, SMAC) automate this search.
- They run multiple *trials*: train the model with different hyperparameters and measure validation score.
- A smart search strategy balances *exploration* (try new regions) and *exploitation* (refine good ones).
- Often includes *early stopping/pruning* to kill bad trials fast and save compute.
- Visualizing optimizers (Ruder blog)