HW4: Optimizers on MNIST – Plots & Implementation

Rishik Saha (2022CS11932)

August 2025

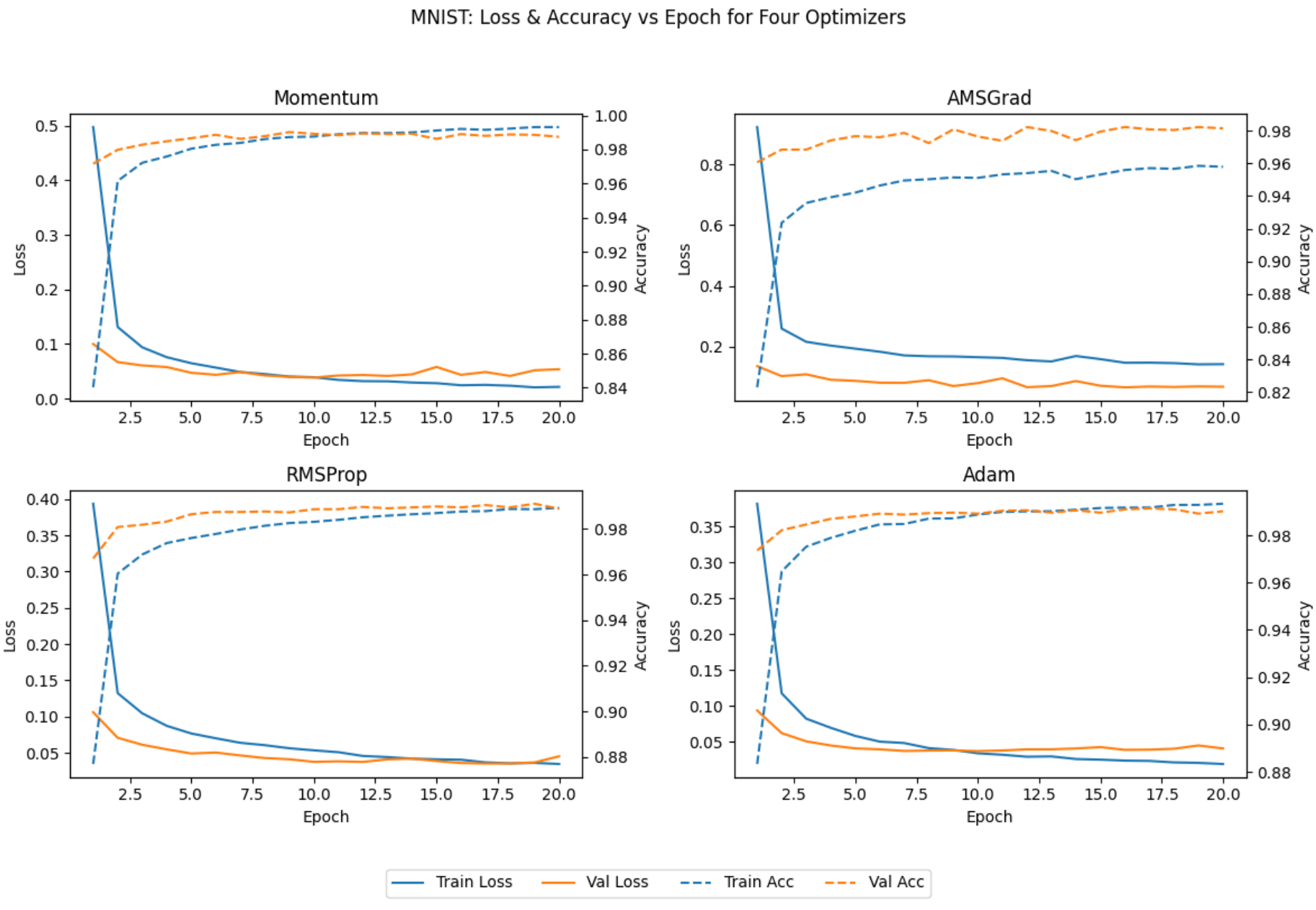

Figure 1: Training & validation accuracy and loss vs. epoch comparing **Momentum**, **AMSGrad**, **RMSProp**, and **Adam**. The Kaggle MNIST IDX dataset is used with the provided fixed hyperparameters and the supplied small CNN.

Optimizer Implementations (from scratch)

The following implementations were used for training: Classical Momentum, AMSGrad, RMSProp, and Adam (with bias correction).

```
class MomentumSGD(BaseOptimizer):
    def __init__(self, params, lr=0.05, momentum=0.9):
        super().__init__(params)
        self.lr = lr
        self.momentum = momentum
        for p in self.params:
            self.state[p] = {"v": torch.zeros_like(p)}

    @torch.no_grad()
    def step(self):
        for p in self.params:
            if p.grad is None:
                continue
            g = p.grad
            st = self.state[p]
            v = st["v"]

            v.mul_(self.momentum).add_(g, alpha=self.lr)

            p.add_(v, alpha=-1.0)

class AMSGrad(BaseOptimizer):
    def __init__(self, params, lr=0.03, beta1=0.9, beta2=0.999, eps=1e-8):
        super().__init__(params)
        self.lr, self.beta1, self.beta2, self.eps = lr, beta1, beta2, eps
        for p in self.params:
            self.state[p] = {
                "t": 0,
                "m": torch.zeros_like(p),
                "v": torch.zeros_like(p),
                "vmax": torch.zeros_like(p)
            }

    @torch.no_grad()
    def step(self):
        b1, b2, eps = self.beta1, self.beta2, self.eps
        for p in self.params:
            if p.grad is None:
                continue
            g = p.grad
            st = self.state[p]
            st["t"] += 1
            t = st["t"]
            m, v, vmax = st["m"], st["v"], st["vmax"]

            m.mul_(b1).add_(g, alpha=1 - b1)
            v.mul_(b2).addcmul_(g, g, value=1 - b2)

            mhat = m / (1 - b1**t)
            vhat = v / (1 - b2**t)

            torch.maximum(vmax, vhat, out=vmax)

            p.addcdiv_(mhat, vmax.sqrt().add_(eps), value=-self.lr)

class RMSProp(BaseOptimizer):
    def __init__(self, params, lr=1e-3, alpha=0.9, eps=1e-8):
        super().__init__(params)
        self.lr = lr
        self.alpha = alpha
        self.eps = eps
        for p in self.params:
            self.state[p] = {"Eg2": torch.zeros_like(p)}

    @torch.no_grad()
    def step(self):
        a, eps = self.alpha, self.eps
        for p in self.params:
            if p.grad is None:
                continue
            g = p.grad
            Eg2 = self.state[p]["Eg2"]

            Eg2.mul_(a).addcmul_(g, g, value=1 - a)

            denom = (Eg2 + eps).sqrt()
            p.addcdiv_(g, denom, value=-self.lr)

class Adam(BaseOptimizer):
    def __init__(self, params, lr=1e-3, beta1=0.9, beta2=0.999, eps=1e-8):
        super().__init__(params)
        self.lr, self.beta1, self.beta2, self.eps = lr, beta1, beta2, eps
        for p in self.params:
            self.state[p] = {"t": 0, "m": torch.zeros_like(p), "v": torch.zeros_like(p)}

    @torch.no_grad()
    def step(self):
        b1, b2, eps = self.beta1, self.beta2, self.eps
        for p in self.params:
            if p.grad is None:
                continue
            g = p.grad
            st = self.state[p]
            st["t"] += 1
            t = st["t"]
            m, v = st["m"], st["v"]

            m.mul_(b1).add_(g, alpha=1 - b1)
            v.mul_(b2).addcmul_(g, g, value=1 - b2)

            mhat = m / (1 - b1**t)
            vhat = v / (1 - b2**t)

            p.addcdiv_(mhat, vhat.sqrt().add_(eps), value=-self.lr)
```