Improving deep networks: initialization, regularization, optimizers, schedules and batch norm

Real experiments, small enough for a laptop CPU: how initialization changes training, what weight decay and dropout do to the train/dev gap, how SGD, momentum, RMSprop, Adam and AdamW compare, learning-rate schedules, batch normalization, mixed precision and torch.compile.

Open in Google Colab · part of the inPace course library. Slides © DeepLearning.AI (CC BY-SA 2.0); this notebook: inPace (CC BY-SA 4.0).

Run it top to bottom (Runtime → Run all in Colab). Each part says what to look for.

```
# Works in Google Colab as it is (PyTorch is already installed there).
# On your own computer:  pip install torch torchvision scikit-learn scipy
matplotlib pandas
# One framework per notebook: PyTorch and TensorFlow can clash when loaded into
the same Python process.
import numpy as np, matplotlib.pyplot as plt, torch
from torch import nn
torch.manual_seed(0); np.random.seed(0)
device = "cuda" if torch.cuda.is_available() else "mps" if
torch.backends.mps.is_available() else "cpu"
print("PyTorch", torch.__version__, "| device:", device)
```

```
PyTorch 2.15.0+cu132 | device: cpu
```

0. Data and a helper

Handwritten digits (8×8), split into train / dev / test. `run()` trains a model and records the losses.

```
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
d = load_digits()
X, Xt_, y, yt_ = train_test_split(d.data / 16.0, d.target, test_size=0.4,
random_state=0)
Xd_, Xte, yd_, yte = train_test_split(Xt_, yt_, test_size=0.5, random_state=0)
T = lambda a, f=torch.float32: torch.tensor(a, dtype=f)
Xtr, ytr, Xdv, ydv, Xts, yts = T(X), T(y, torch.long), T(Xd_), T(yd_, torch.long),
T(Xte), T(yte, torch.long)
print("train", tuple(Xtr.shape), "dev", tuple(Xdv.shape), "test",
tuple(Xts.shape))
loader = torch.utils.data.DataLoader(torch.utils.data.TensorDataset(Xtr, ytr),
batch_size=64, shuffle=True)

def accuracy(m, X, y):
    m.eval()
    with torch.no_grad(): return (m(X).argmax(1) == y).float().mean().item()
```

```
def run(model, opt, epochs=40, sched=None):
    hist = {"train": [], "dev": []}
    for _ in range(epochs):
        model.train()
        for xb, yb in loader:                       # mini-batch gradient descent
            opt.zero_grad(); nn.functional.cross_entropy(model(xb),
yb).backward(); opt.step()
            if sched: sched.step()
        model.eval()
        with torch.no_grad():
            hist["train"].append(nn.functional.cross_entropy(model(Xtr),
ytr).item())
            hist["dev"].append(nn.functional.cross_entropy(model(Xdv),
ydv).item())
    return hist

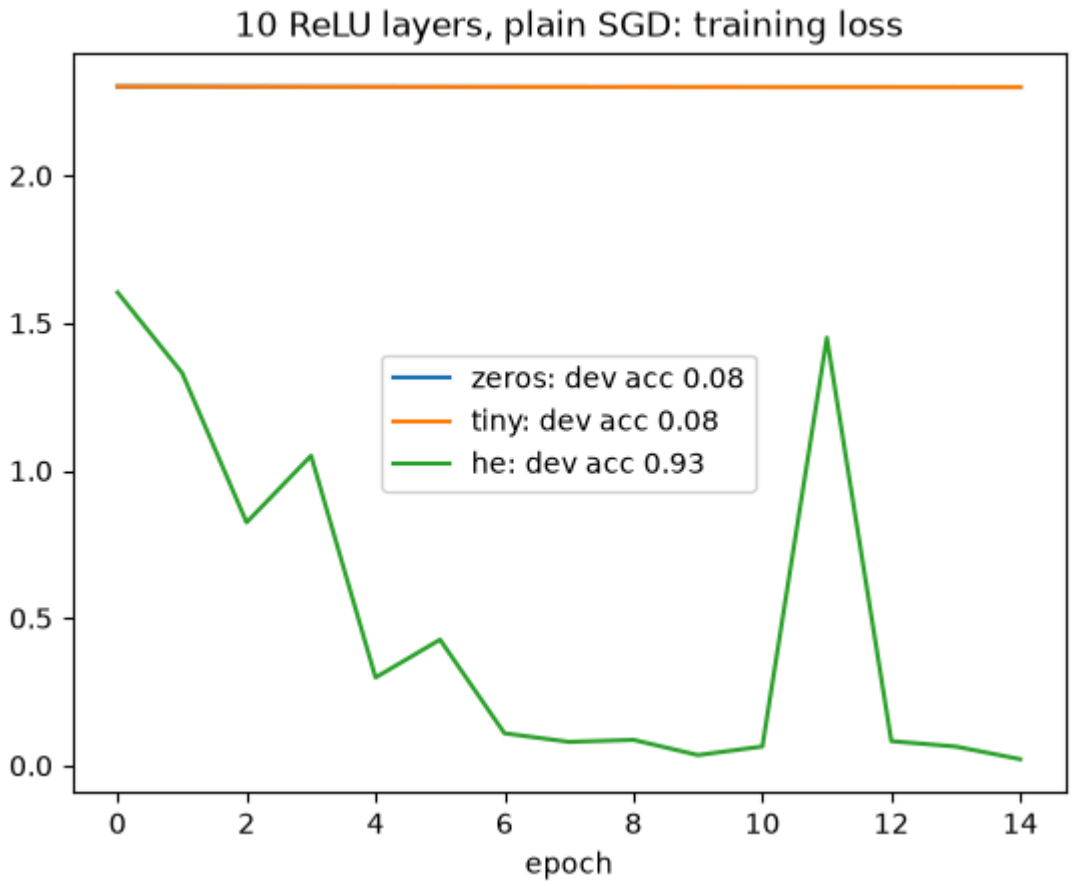
def mlp(width=256, depth=4, dropout=0.0, bn=False):
    layers, n = [], 64
    for _ in range(depth):
        layers += [nn.Linear(n, width)] + ([nn.BatchNorm1d(width)] if bn else [])
+ [nn.ReLU(), nn.Dropout(dropout)]
        n = width
    return nn.Sequential(*layers, nn.Linear(n, 10))
```

```
train (1078, 64) dev (359, 64) test (360, 64)
```

1. Initialization

PyTorch's `nn.Linear` already uses a sensible default (Kaiming-uniform). Here we force three choices on a 10-layer network and watch the first epochs.

```
def init_with(model, how):
    for m in model.modules():
        if isinstance(m, nn.Linear):
            if how == "zeros": nn.init.zeros_(m.weight)
            elif how == "tiny": nn.init.normal_(m.weight, std=0.01)
            elif how == "he": nn.init.kaiming_normal_(m.weight,
nonlinearity="relu")
            nn.init.zeros_(m.bias)
    return model
for how in ("zeros", "tiny", "he"):
    torch.manual_seed(0)
    m = init_with(mlp(width=128, depth=10), how)
    h = run(m, torch.optim.SGD(m.parameters(), lr=0.05), epochs=15)
    plt.plot(h["train"], label=f"{how}: dev acc {accuracy(m, Xdv, ydv):.2f}")
plt.legend(); plt.title("10 ReLU layers, plain SGD: training loss");
plt.xlabel("epoch"); plt.show()
```



2. Regularization: weight decay and dropout

We train on only **150 examples** so the network can memorize them. Look at the gap between train and dev.

```
small = torch.utils.data.TensorDataset(Xtr[:150], ytr[:150])
loader_full, loader = loader, torch.utils.data.DataLoader(small, batch_size=50,
shuffle=True)
rows = []
for name, kw, wd in (("nothing", {}, 0.0), ("weight decay 1.0", {}, 1.0),
("dropout 0.5", {"dropout": 0.5}, 0.0)):
    torch.manual_seed(0)
    m = mlp(512, 3, **kw)
    h = run(m, torch.optim.AdamW(m.parameters(), lr=1e-3, weight_decay=wd),
epochs=200)
    rows.append((name, accuracy(m, Xtr[:150], ytr[:150]), accuracy(m, Xdv, ydv)))
    plt.plot(h["dev"], label=f"{name} (dev loss)")
loader = loader_full
plt.legend(); plt.ylim(0, 1.5); plt.title("dev loss with 150 training examples");
plt.xlabel("epoch"); plt.show()
for r in rows: print(f"{r[0]:<18} train acc {r[1]:.3f} | dev acc {r[2]:.3f}")
```

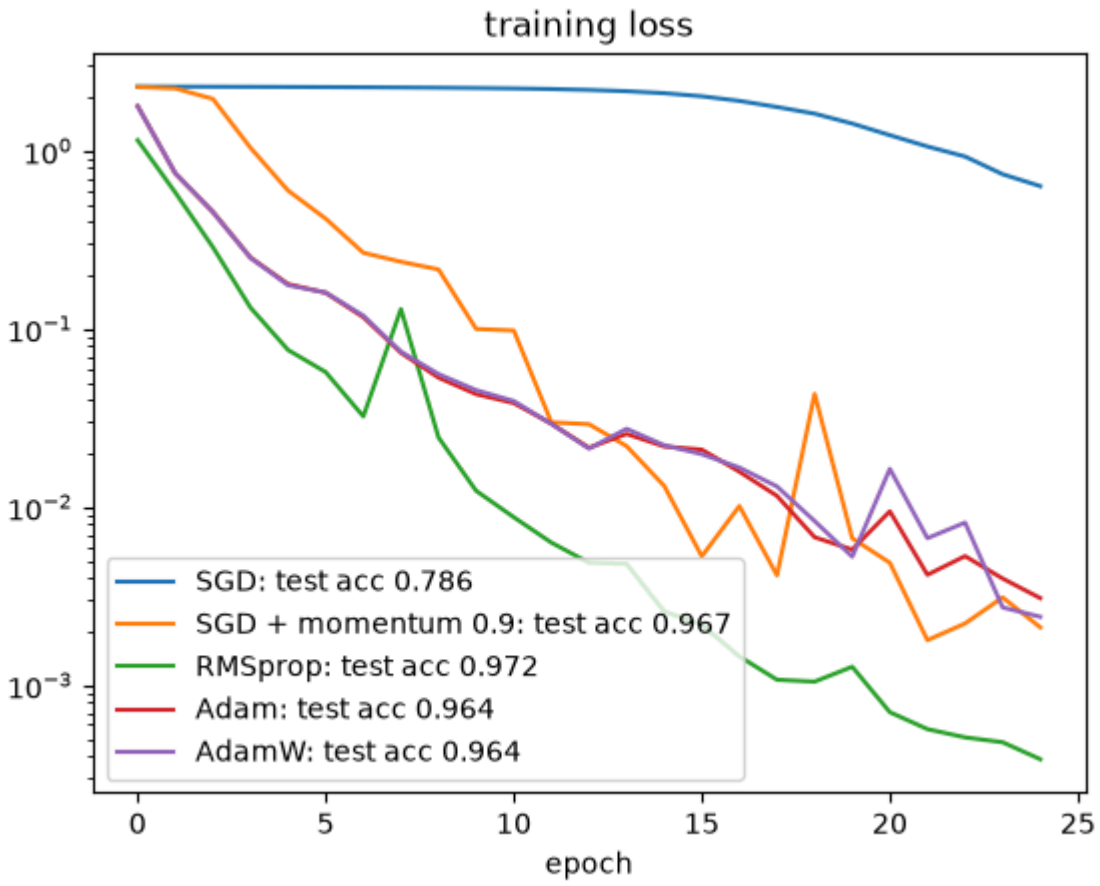



```
nothing             train acc 1.000 | dev acc 0.903
weight decay 1.0    train acc 1.000 | dev acc 0.916
dropout 0.5         train acc 1.000 | dev acc 0.936
```

3. Optimizers

Same network, same mini-batches, five optimizers. `AdamW` is Adam with *decoupled* weight decay: the modern default for most networks.

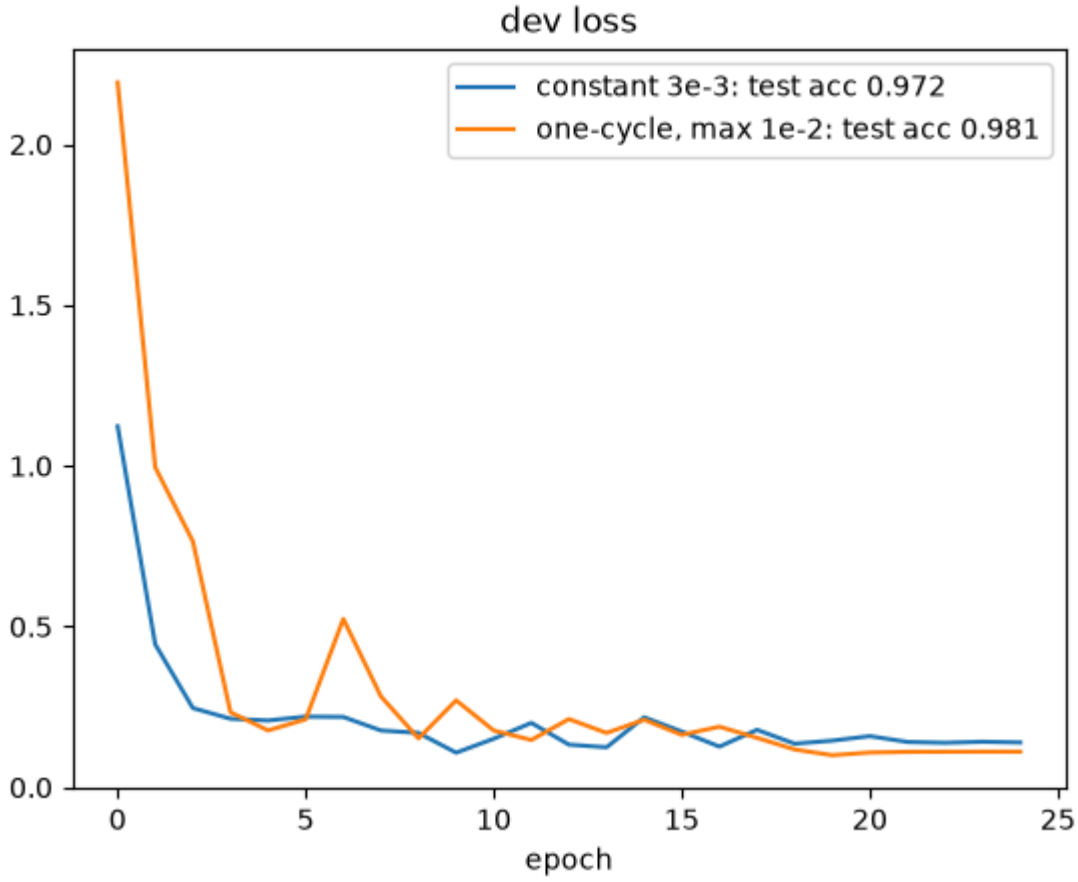
```
opts = {"SGD": lambda p: torch.optim.SGD(p, lr=0.05),
        "SGD + momentum 0.9": lambda p: torch.optim.SGD(p, lr=0.05, momentum=0.9),
        "RMSprop": lambda p: torch.optim.RMSprop(p, lr=1e-3),
        "Adam": lambda p: torch.optim.Adam(p, lr=1e-3),
        "AdamW": lambda p: torch.optim.AdamW(p, lr=1e-3, weight_decay=0.01)}
for name, make in opts.items():
    torch.manual_seed(0); m = mlp(256, 4)
    h = run(m, make(m.parameters()), epochs=25)
    plt.plot(h["train"], label=f"{name}: test acc {accuracy(m, Xts, yts):.3f}")
plt.yscale("log"); plt.legend(); plt.title("training loss"); plt.xlabel("epoch");
plt.show()
```



4. Learning-rate schedules

A constant learning rate vs **one-cycle** (warm up, then anneal). The scheduler steps after every mini-batch.

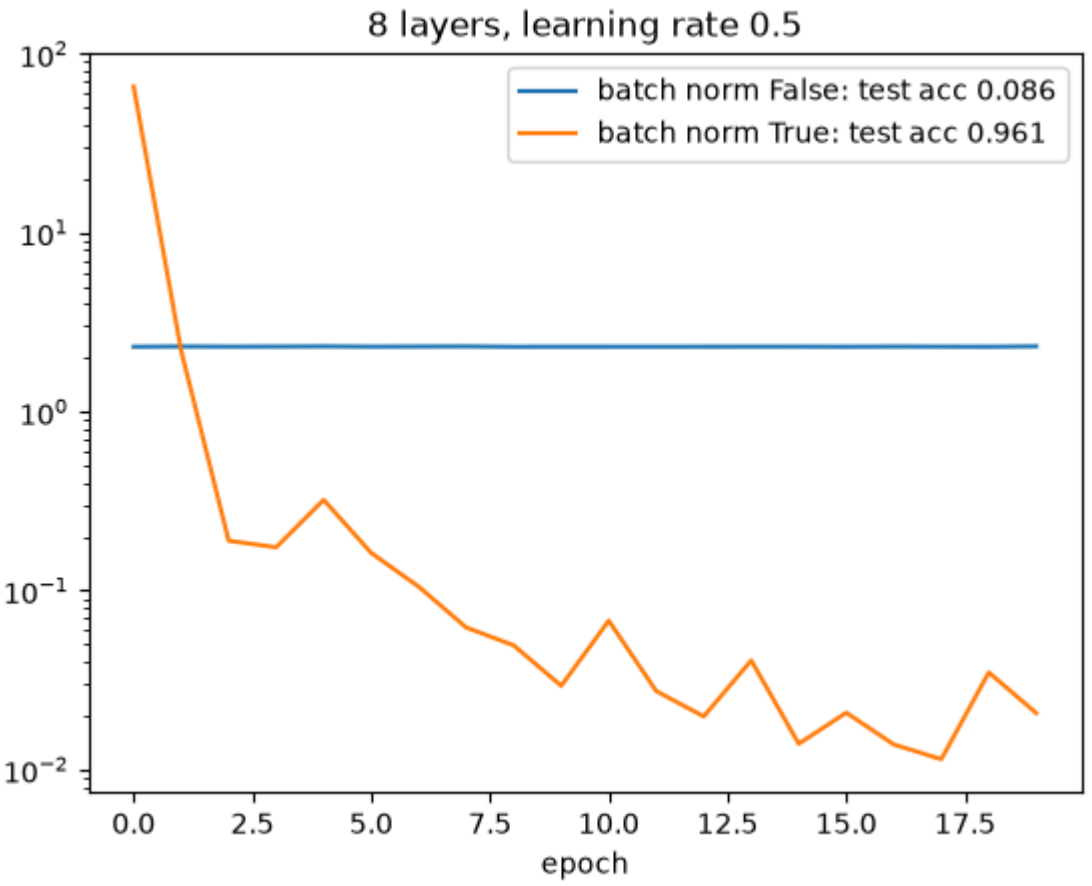
```
for name in ("constant 3e-3", "one-cycle, max 1e-2"):
    torch.manual_seed(0); m = mlp(256, 4)
    opt = torch.optim.AdamW(m.parameters(), lr=3e-3)
    sched = torch.optim.lr_scheduler.OneCycleLR(opt, max_lr=1e-2, total_steps=25 *
len(loader)) if "one" in name else None
    h = run(m, opt, epochs=25, sched=sched)
    plt.plot(h["dev"], label=f"{name}: test acc {accuracy(m, Xts, yts):.3f}")
plt.legend(); plt.title("dev loss"); plt.xlabel("epoch"); plt.show()
```



5. Batch normalization

With a learning rate that's too high for plain layers, batch norm keeps each layer's inputs in a steady range.

```
for bn in (False, True):
    torch.manual_seed(0); m = mlp(256, 8, bn=bn)
    h = run(m, torch.optim.SGD(m.parameters(), lr=0.5, momentum=0.9), epochs=20)
    plt.plot(h["train"], label=f"batch norm {bn}: test acc {accuracy(m, Xts,
yts):.3f}")
plt.yscale("log"); plt.legend(); plt.title("8 layers, learning rate 0.5");
plt.xlabel("epoch"); plt.show()
```



6. Check gradients the PyTorch way

`torch.autograd.gradcheck` compares autograd with numerical differences (use float64).

```
layer = nn.Sequential(nn.Linear(4, 3), nn.Tanh()).double()
x = torch.randn(5, 4, dtype=torch.float64, requires_grad=True)
print("gradcheck passed:", torch.autograd.gradcheck(lambda t: layer(t).sum(),
(x,)))
```

```
gradcheck passed: True
```

7. Faster training on a GPU: mixed precision and torch.compile

On a GPU (Colab: Runtime → Change runtime type → T4 GPU), `autocast` runs most maths in 16-bit floats and `torch.compile` fuses operations. On a CPU this cell just runs normally.

```
m = mlp(256, 4).to(device)
opt = torch.optim.AdamW(m.parameters(), lr=1e-3)
use_amp = device == "cuda"
fast = torch.compile(m) if device == "cuda" else m      # compiling pays off on
GPUs and long runs
for xb, yb in loader:
    xb, yb = xb.to(device), yb.to(device)
    with torch.autocast(device_type="cuda" if use_amp else "cpu",
dtype=torch.bfloat16, enabled=use_amp):
        loss = nn.functional.cross_entropy(fast(xb), yb)
```

```
    opt.zero_grad(); loss.backward(); opt.step()
print("one epoch done on", device, "| last loss", round(loss.item(), 3))
```

```
one epoch done on cpu | last loss 1.701
```

Try this

1. In section 2, try weight decay 0.01, 0.1, 1.0 and 3.0. Plot dev accuracy against weight decay.
2. In section 3, give plain SGD a 10× bigger learning rate. Does it catch up, or diverge?
3. In section 5, lower the learning rate to 0.05. Does batch norm still matter as much?
4. Random search: pick 10 random learning rates on a log scale between 1e-4 and 1e-1 (`10 ** np.random.uniform(-4, -1)`) and keep the best by dev loss.