

NEURAL NET TRAINING

01 ACTIVATION FUNCTIONS

	FORMULA - DERIVATIVE	NUMPY	PYTORCH
Sigmoid	$\sigma(x) = 1 / (1 + e^{-x})$ $\sigma' = \sigma(1 - \sigma)$	<code>1/(1+np.exp(-x))</code>	<code>torch.sigmoid(x)</code>
Tanh	$(e^x - e^{-x}) / (e^x + e^{-x})$ $\tanh' = 1 - \tanh^2$	<code>np.tanh(x)</code>	<code>torch.tanh(x)</code>
ReLU	$\max(0, x)$ $\text{ReLU}' = 1$ if $x > 0$ else 0	<code>np.maximum(0, x)</code>	<code>F.relu(x)</code>
Leaky ReLU	x if $x > 0$ else αx , $\alpha \approx 0.01$	<code>np.where(x>0, x, a*x)</code>	<code>F.leaky_relu(x)</code>
GELU	$x \cdot \Phi(x) \approx \frac{1}{2}x(1 + \tanh[\sqrt{2/\pi}(x + 0.0447x^3)])$	<code>.5*x*(1+np.tanh(.7979*(x+.0447*x**3)))</code>	<code>F.gelu(x)</code>
SILU / Swish	$x \cdot \sigma(x)$, used in Llama	<code>x/(1+np.exp(-x))</code>	<code>F.silu(x)</code>
Softmax	$e^{x_i} / \sum_j e^{x_j}$ subtract $\max(x)$ for stability	<code>e = np.exp(x-x.max())</code> <code>e / e.sum()</code>	<code>F.softmax(x, dim=-1)</code>

02 LOSS FUNCTIONS

	FORMULA (MEAN OVER N)	NUMPY	PYTORCH
MSE (L2)	$1/n \sum (y - \hat{y})^2$	<code>np.mean((y-p)**2)</code>	<code>F.mse_loss(p, y)</code>
MAE (L1)	$1/n \sum y - \hat{y} $, robust to outliers	<code>np.mean(np.abs(y-p))</code>	<code>F.l1_loss(p, y)</code>
Huber	$\frac{1}{2}\epsilon^2$ if $ e \leq \delta$ else $\delta(e - \frac{1}{2}\delta)$	<code>a = np.abs(y-p)</code> <code>np.mean(np.where(a<=d, .5*a**2, d*(a-.5*d)))</code>	<code>F.huber_loss(p, y)</code>
Binary CE	$-[y \log p + (1 - y) \log(1 - p)]$ $p = \sigma(z)$; $\partial L / \partial z = p - y$	<code>-np.mean(y*np.log(p) + (1-y)*np.log(1-p))</code>	<code>F.binary_cross_entropy_with_logits(z, y)</code>
Cross-entropy	$-\log \text{softmax}(z)_y = -z_y + \log \sum_j e^{z_j}$ $\partial L / \partial z = \text{softmax}(z) - \text{onehot}(y)$	<code>-np.log(P[range(n),y]).mean()</code>	<code>F.cross_entropy(z, y)</code>
KL div.	$\sum p \log(p / q)$, distillation	<code>np.sum(p*np.log(p/q))</code>	<code>F.kl_div(q.log(), p, reduction="batchmean")</code>

PyTorch's cross_entropy and BCE...with_logits take raw logits (no softmax / sigmoid first): they're fused and numerically stable.

TRAINING CHECKLIST

Normalize inputs	zero mean, unit variance (train stats only)
Check initial loss	$\approx \ln K$ for K balanced classes; NaN $\rightarrow$ lower lr, clip grads
Overfit one batch	loss $\rightarrow 0$, else there's a bug
train() / eval()	toggle dropout & BatchNorm
Train $\downarrow$ val $\uparrow$	overfit: dropout, decay, more data, early stop

AUTOGRAD & GRADIENTS

Chain rule	$\partial L / \partial x = \partial L / \partial y \cdot \partial y / \partial x$, layer by layer	<code>loss.backward(); w.grad</code>
Grad check	$f'(x) \approx (f(x+h) - f(x-h)) / 2h$, $h \approx 1e-5$	<code>torch.autograd.gradcheck(f, x)</code>
No grad	stop tracking (eval, frozen parts)	<code>torch.no_grad(), x.detach()</code>

03 OPTIMIZERS

	UPDATE RULE ($g = \nabla_g L$, η = learning rate)	TORCH.OPTIM
SGD	$\theta \leftarrow \theta - \eta \cdot g$	<code>SGD(params, lr=0.1)</code>
Momentum	$v \leftarrow \mu v + g$; $\theta \leftarrow \theta - \eta \cdot v$	<code>SGD(..., momentum=0.9)</code>
RMSProp	$s \leftarrow \rho s + (1 - \rho)g^2$; $\theta \leftarrow \theta - \eta \cdot g / (\sqrt{s} + \epsilon)$	<code>RMSprop(..., alpha=0.99)</code>
Adam	$m \leftarrow \beta_1 m + (1 - \beta_1)g$; $\hat{v} \leftarrow \beta_2 v + (1 - \beta_2)g^2$ $\hat{m} = m / (1 - \beta_1^t)$, $\hat{v} = \hat{v} / (1 - \beta_2^t)$; $\theta \leftarrow \theta - \eta \cdot \hat{m} / (\sqrt{\hat{v}} + \epsilon)$	<code>Adam(..., lr=1e-3, betas=(0.9, 0.999))</code>
AdamW	Adam + decoupled decay $\theta \leftarrow \theta - \eta \lambda \theta$	<code>AdamW(..., weight_decay=0.01)</code>

```
1 def adamw_step(w, g, m, v, t, lr=1e-3, b1=0.9, b2=0.999,
2               eps=1e-8, wd=0.01):
3     # NumPy, t starts at 1
4     m = b1 * m + (1 - b1) * g          # 1st moment (mean)
5     v = b2 * v + (1 - b2) * g ** 2      # 2nd moment (variance)
6     m_hat, v_hat = m / (1 - b1 ** t), v / (1 - b2 ** t) # bias fix
7     w = w - lr * (m_hat / (np.sqrt(v_hat) + eps) + wd * w)
8     return w, m, v
```

LEARNING-RATE SCHEDULES (TORCH.OPTIM.LR.SCHEDULER)

StepLR	$\eta \cdot \gamma^{\lfloor \text{epoch} / s \rfloor}$	<code>StepLR(opt, 30, gamma=0.1)</code>
Cosine	$\eta_{\min} + \frac{1}{2}(\eta_{\max} - \eta_{\min})(1 + \cos(\pi t / T))$	<code>CosineAnnealingLR(opt, T_max=T)</code>
Warmup	linear ramp $\sim 0 \rightarrow \eta$ over first steps	<code>LinearLR(opt, 0.01, total_iters=500)</code>
OneCycle	warm up to $\max_lr$, then anneal	<code>OneCycleLR(opt, max_lr, total_steps=N)</code>
On plateau	$\times$ factor when the val metric stalls	<code>ReduceLROnPlateau(opt, patience=5)</code>

04 METRICS

	FORMULA	NUMPY	PYTORCH
Accuracy	$(TP + TN) / N$	<code>np.mean(pred == y)</code>	<code>(pred==y).float().mean()</code>
Precision	$TP / (TP + FP)$, predicted + that are right	<code>p1, y1 = pred==1, y==1</code> <code>tp = (p1 & y1).sum()</code> <code>fp = (p1 & ~y1).sum()</code>	<code>torchmetrics.F1Score(task="multiclass", num_classes=K)</code>
Recall	$TP / (TP + FN)$, real + that are found	<code>fn = (~p1 & y1).sum()</code> <code>P, R = tp/(tp+fp), tp/(tp+fn)</code>	
F1	$2PR / (P + R)$, for imbalanced data	<code>f1 = 2*P*R / (P + R)</code>	
Confusion	$C[i, j] = \#(\text{true } i, \text{ predicted } j)$	<code>C=np.zeros((K,K),int)</code> <code>np.add.at(C,(y,pred),1)</code>	<code>torchmetrics.ConfusionMatrix</code>
ROC-AUC	area under the TPR-vs-FPR curve	<code>roc_auc_score(y, s)</code> (sklearn)	<code>torchmetrics.AUROC</code>
R²	$1 - \Sigma(y - \hat{y})^2 / \Sigma(y - \bar{y})^2$	<code>1 - ((y-p)**2).sum() / ((y-y.mean())**2).sum()</code>	<code>torchmetrics.R2Score</code>
Perplexity	$\exp(\text{mean cross-entropy})$	<code>np.exp(ce_loss)</code>	<code>torch.exp(loss)</code>

05 INIT, NORM & REGULARIZATION

He / Kaiming	$W \sim N(0, 2 / n_{\text{in}})$, for ReLU nets	<code>nn.init.kaiming_normal_(w)</code>
Xavier	$\text{Var} = 2 / (n_{\text{in}} + n_{\text{out}})$, tanh / sigmoid	<code>nn.init.xavier_uniform_(w)</code>
BatchNorm	$\gamma(x - \mu_B) / \sqrt{(\sigma_B^2 + \epsilon)} + \beta$, stats over batch	<code>nn.BatchNorm1d(n)</code>
LayerNorm	same, stats over features of each sample	<code>nn.LayerNorm(n)</code>
Dropout	zero with prob p , scale rest by $1 / (1 - p)$	<code>nn.Dropout(p=0.1)</code>
Weight decay	$L + \lambda \ w\ ^2$, or $\theta \leftarrow \theta - \eta \lambda \theta$ (decoupled)	<code>AdamW(weight_decay=0.01)</code>

06 MLP FROM SCRATCH (NUMPY)

Backprop is the chain rule for a dense layer $Z = XW + b$: $dW = X^T \cdot dz$, $db = \Sigma dz$, $dX = dZ \cdot W^T$.

```
1 import numpy as np
2 def softmax(z):
3     e = np.exp(z - z.max(1, keepdims=True)) # stable
4     return e / e.sum(1, keepdims=True)
5
6 rng = np.random.default_rng(0) # X: [N, D], y: [N]
7 W1 = rng.normal(0, np.sqrt(2 / D), (D, H)); b1 = np.zeros(H)
8 W2 = rng.normal(0, np.sqrt(2 / H), (H, K)); b2 = np.zeros(K)
9 for epoch in range(2000):
10     Z1 = X @ W1 + b1; A1 = np.maximum(0, Z1) # ReLU
11     P = softmax(A1 @ W2 + b2)
12     loss = -np.log(P[np.arange(N), y]).mean()
13     dZ2 = P.copy(); dZ2[np.arange(N), y] -= 1 # dL/dz
14     dZ2 /= N # softmax-onehot
15     dW2, db2 = A1.T @ dZ2, dZ2.sum(0)
16     dZ1 = (dZ2 @ W2.T) * (Z1 > 0) # ReLU'
17     dW1, db1 = X.T @ dZ1, dZ1.sum(0)
18     for p, g in [(W1, dW1), (b1, db1), (W2, dW2), (b2, db2)]:
19         p -= lr * g # SGD step
```

07 SAME IN PYTORCH

```
1 model = nn.Sequential(nn.Linear(D, H), nn.ReLU(),
2                       nn.Dropout(0.1), nn.Linear(H, K)).to(dev)
3 loss_fn = nn.CrossEntropyLoss()
4 opt = torch.optim.AdamW(model.parameters(), lr=1e-3)
5 sched = torch.optim.lr_scheduler.OneCycleLR(
6     opt, max_lr=1e-3, total_steps=E * len(train_dl))
7 for epoch in range(E):
8     model.train() # dropout / BN on
9     for xb, yb in train_dl:
10         xb, yb = xb.to(dev), yb.to(dev)
11         loss = loss_fn(model(xb), yb) # logits in
12         opt.zero_grad() # grads accumulate!
13         loss.backward() # autograd = backprop
14         nn.utils.clip_grad_norm_(model.parameters(), 1.0)
15         opt.step(); sched.step()
16     model.eval(); correct = 0
17     with torch.no_grad():
18         for xb, yb in val_dl:
19             pred = model(xb.to(dev)).argmax(1)
20             correct += (pred == yb.to(dev)).sum().item()
21 acc = correct / len(val_dl.dataset)
```