LECTURE 9 CHEATSHEET

Make invalid outputs unreachable

Learn unrestricted coordinates; transform them into valid model parameters.

$z \in \mathbb{R}^d \to \theta = \varphi(z) \in \mathcal{C}$: choose the map from the parameter's domain.

1 · Constraint map

Need	Raw	Transform
positive scale	$u \in \mathbb{R}$	$\sigma = \mathrm{softplus}(u) + \varepsilon$
probability	$z \in \mathbb{R}$	$p = \mathrm{sigmoid}(z)$
simplex	$\boldsymbol{z} \in \mathbb{R}^K$	$\boldsymbol{p} = \mathrm{softmax}(\boldsymbol{z})$
PD covariance	raw triangle	positive-diagonal L; $\Sigma = LL^T$

The optimizer never sees a constrained coordinate directly; the chain rule carries gradients through φ.

2 · Positive scales

$$\mathrm{softplus}(u) = \log(1 + e^u) > 0.$$

$$\mathrm{softplus}'(u) = \mathrm{sigmoid}(u).$$

Use $\sigma = \mathrm{softplus}(u) + \varepsilon$ for a strictly positive numerical floor. Compared with e^u, softplus grows approximately linearly for large positive u, so raw steps are gentler.

Extremely negative raw scales can still learn slowly because the sigmoid derivative approaches zero. Initialize sensibly.

3 · Binary probabilities

$$p = \mathrm{sigmoid}(z) = \tfrac{1}{1+e^{-z}}, \quad p \in (0, 1).$$

For binary cross-entropy,

$$\frac{\partial \ell}{\partial z} = p - y.$$

Train from raw logits with a fused primitive; compute probabilities only for interpretation.

```
loss = F.binary_cross_entropy_with_logits(z, y)
p = z.sigmoid()  # reporting
```

Shape guard

Assert `y.shape == logits.shape` for elementwise binary loss. `[N]` and `[N,1]` can broadcast into an unintended `[N,N]` calculation.

4 · Multiclass probabilities

$$p_k = \tfrac{e^{z_k}}{\sum_j e^{z_j}}, \quad p_k > 0, \quad \textstyle\sum_k p_k = 1.$$

Adding the same constant to every logit leaves probabilities unchanged. For stable computation, subtract $m = \max_j z_j$:

$$p_k = \tfrac{e^{z_k - m}}{\sum_j e^{z_j - m}}.$$

Since $z_j - m \le 0$, no exponential overflows and the denominator contains at least one $e^0 = 1$.

5 · Stay in log space for training

Stable cross-entropy for target class t:

$$\ell(\boldsymbol{z}, t) = -(z_t - m) + \log\left(\sum_j e^{z_j - m}\right).$$

```
# logits: [N, K], target: [N] integer classes
loss = F.cross_entropy(logits, target)
log_p = F.log_softmax(logits, dim=-1)
```

Do not compute `log(softmax(logits))` as two separate operations. Stability cannot repair `NaN` or infinite input logits.

6 · Positive-definite covariance

Decode raw outputs into a lower-triangular factor L. Make each diagonal entry positive, then set

$$\Sigma = LL^T.$$

For any nonzero vector $\boldsymbol{a}$,

$$\boldsymbol{a}^T \Sigma \boldsymbol{a} = \left\| L^T \boldsymbol{a} \right\|^2 > 0$$

when L is nonsingular. In PyTorch distributions, pass the factor directly when a `scale_tril` interface is available.

7 · Domain audit

- ✓ Write the legal parameter set before choosing the output head.
- ✓ Keep the raw output unrestricted.
- ✓ Use a differentiable map with a useful slope.
- ✓ Use fused stable losses on logits.
- ✓ Assert shapes, dtypes, and finite values.

LECTURE 9 CHEATSHEET

Schedule the learning rate

Large steps help early; smaller steps give noisy optimization more control near a minimum.

$\theta_{t+1} = \theta_t - \eta_t d_t$: **the optimizer builds d_t; the schedule chooses its scalar scale.**

1 · Why decay helps

A minibatch gradient can be written

$$g_{\mathcal{B}}(\theta) = \nabla \mathcal{L}(\theta) + \varepsilon_{\mathcal{B}}.$$

Near a minimum, the true gradient becomes small while minibatch noise remains. The update noise has scale approximately $\eta_t \varepsilon_{\mathcal{B}}$, so shrinking η_t reduces the bounce.

With exact full-batch gradients, decay is still useful for finer late-stage steps; the noise argument is strongest for SGD/minibatches.

2 · Four useful time-based rules

Step: $\eta_t = \eta_0 \gamma^{\lfloor \frac{t}{S} \rfloor}$
Exponential: $\eta_t = \eta_0 \gamma^t$
Inverse-time: $\eta_t = \frac{\eta_0}{1+kt}$
Cosine: $\eta_t = \eta_{\min} + \frac{1}{2}(\eta_{\max} - \eta_{\min})(1 + \cos(\pi \frac{t}{T}))$

Define the clock: t may mean update, epoch, or scheduler call. The formula is incomplete without that convention.

3 · Warm up, then decay

During the first W updates,

$$\eta_t = \eta_{\text{peak}} \frac{t}{W}.$$

Then switch to the chosen decay rule. Warmup is useful when startup at the peak rate causes loss spikes, divergence, or unstable large updates.

Warmup is not mandatory. Add it to solve an observed startup problem, and count it in the total update budget.

Fixed budget

Cosine decay is a strong simple default when the total number of updates is known. Choose the endpoint and minimum rate explicitly.

4 · Time-based or metric-based?

Situation	Simple choice
fixed run length	cosine or a predeclared step schedule
validation decides	reduce on plateau after validation
fragile startup	warmup, then time-based decay
short exploratory run	constant rate for interpretability

A plateau scheduler consumes a validation metric; it is part of model selection and must never watch the test set.

5 · Correct call order

```
for x, y in train_loader:
    opt.zero_grad(set_to_none=True)
    loss = criterion(model(x), y)
    loss.backward()
    opt.step()

sched.step()  # if the clock is one epoch
```

For `ReduceLROnPlateau`, evaluate first and call `sched.step(val_loss)`. Log the rate used for each update.

6 · Schedule audit

- ✓ The optimizer's base learning rate was tuned first.
- ✓ The schedule clock and total horizon are explicit.
- ✓ Warmup and decay share one continuous rate trace.
- ✓ Checkpoint resume restores optimizer, scheduler, and clock.
- ✓ Comparisons use the same update budget.
- ✓ Validation—not test—chooses metric-based changes.

Choose the simplest schedule that matches the run; add complexity only for a diagnosed failure mode.