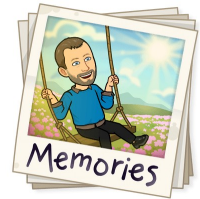


Machine Learning

Textbook: Machine Learning Refined, by Watt, Borhani, and Katsaggelos, 2nd edition:
MLRefined.com

Previous Lecture

- ◆ Backpropagation is repeated application of the chain rule from a scalar loss backward through a neural network.
- ◆ For a dense layer, $\nabla_{W_\ell} L = \delta_\ell \mathbf{h}_{\ell-1}^T$ and $\nabla_{\mathbf{b}_\ell} L = \delta_\ell$.
- ◆ Hidden-layer errors satisfy $\delta_\ell = (W_{\ell+1}^T \delta_{\ell+1}) \odot \sigma'_\ell(\mathbf{a}_\ell)$.
- ◆ Regression, sigmoid cross-entropy, and softmax cross-entropy mainly differ in the output-layer error that begins the backward pass.
- ◆ The forward pass caches quantities that are reused by the backward pass, making reverse-mode differentiation efficient.
- ◆ Finite differences are valuable for gradient checking but are too expensive to replace backpropagation during training.



§13.5, 13.7 - Training Neural Networks

Last time:

backpropagation gave

$$\nabla_{\theta} L$$

for every weight and bias.

Now:

use those derivatives

to actually fit the network.

Training requires more than
“compute a gradient.”

We must choose:

initialization;

batch size;

learning rate;

number of epochs;

regularization;

stopping rules.

Today:

assemble the complete training procedure.

Collect All Parameters

Let θ denote every trainable parameter in the network.

For example:
 $\theta = \{W_1, \mathbf{b}_1, \dots, W_L, \mathbf{b}_L, W_{\text{out}}, \mathbf{b}_{\text{out}}\}.$

Training means solving
 $\theta^* = \underset{\theta}{\operatorname{argmin}} \ g(\theta).$

Because hidden features depend on the same parameters being optimized:
 $g(\theta)$
is generally nonconvex.

Backpropagation does not remove this nonconvexity.

It makes the gradient computationally available.

The Dataset Objective

For observations
 $(\mathbf{x}_p, y_p),$
 $p = 1, \dots, P,$

write one-observation loss
 $L_p(\theta).$

The training objective is the finite sum
 $g(\theta) = \frac{1}{P} \sum_{p=1}^P L_p(\theta).$

Thus
 $\nabla g(\theta) = \frac{1}{P} \sum_{p=1}^P \nabla L_p(\theta).$

This is the same finite-sum structure
from Lesson 1.21.

Neural-network training therefore connects directly
to full-batch,
stochastic,
and mini-batch optimization.

Gradient Descent Training

Full-batch gradient descent:
 $\theta_{k+1} = \theta_k - \alpha_k \nabla g(\theta_k).$

At every iteration:

1. run the network forward;

- 2. compute the loss;
- 3. backpropagate;
- 4. update all parameters.

For a large dataset:
 computing the full gradient before every update
 may be unnecessarily expensive.

So neural networks are usually trained
 with mini-batches.

Mini-Batch Training

For mini-batch

$$B \subseteq \{1, \dots, P\}$$

of size

b :

$$g_B(\theta) = \frac{1}{b} \sum_{p \in B} L_p(\theta).$$

Mini-batch gradient:

$$\nabla g_B(\theta) = \frac{1}{b} \sum_{p \in B} \nabla L_p(\theta).$$

Update:

$$\theta \leftarrow \theta - \alpha \nabla g_B(\theta).$$

Backpropagation computes
 the mini-batch gradient
 using matrix operations
 across all observations in the batch.

The Basic Training Loop

A practical epoch:

- 1. shuffle training observations;
- 2. divide them into mini-batches;
- 3. for each mini-batch:
 - forward pass;
 - compute mini-batch loss;
 - backpropagate;
 - update parameters;
- 4. record training diagnostics;
- 5. evaluate validation performance periodically.

Then repeat for another epoch.

One epoch means:
approximately one pass through the training observations.

One Gradient Step From Lesson 1.33

Reuse the one-observation gradients:

$$\nabla_W L = \begin{bmatrix} -4 & -2 \\ 2 & 1 \end{bmatrix},$$

$$\nabla_{\mathbf{b}} L = \begin{bmatrix} -2 \\ 1 \end{bmatrix},$$

$$\nabla_{\mathbf{v}} L = \begin{bmatrix} -1 \\ -3/2 \end{bmatrix},$$

$$\frac{\partial L}{\partial c} = -1.$$

Initial parameters:

$$W = \begin{bmatrix} 1 & -1 \\ 1/2 & 1 \end{bmatrix},$$

$$\mathbf{b} = (0, -1/2)^T,$$

$$\mathbf{v} = (2, -1)^T,$$

$$c = 1/4.$$

Initial loss:

$$L = 0.5.$$

Use Learning Rate $\alpha = 0.01$

Update every parameter:

$$W_{\text{new}} = W - \alpha \nabla_W L,$$

$$\mathbf{b}_{\text{new}} = \mathbf{b} - \alpha \nabla_{\mathbf{b}} L,$$

$$\mathbf{v}_{\text{new}} = \mathbf{v} - \alpha \nabla_{\mathbf{v}} L,$$

$$c_{\text{new}} = c - \alpha \frac{\partial L}{\partial c}.$$

This gives

$$W_{\text{new}} = \begin{bmatrix} 1.04 & -0.98 \\ 0.48 & 0.99 \end{bmatrix},$$

$$\mathbf{b}_{\text{new}} = \begin{bmatrix} 0.02 \\ -0.51 \end{bmatrix},$$

$$\mathbf{v}_{\text{new}} = \begin{bmatrix} 2.01 \\ -0.985 \end{bmatrix},$$

$$c_{\text{new}} = 0.26.$$

Evaluate the Updated Network

For the same input

$$\mathbf{x} = (2, 1)^T:$$

new pre-activations:

$$\mathbf{a}_{\text{new}} = \begin{bmatrix} 1.12 \\ 1.44 \end{bmatrix}.$$

Both remain positive,
so ReLU leaves them unchanged.

New prediction:

$$\hat{y}_{\text{new}} = 1.0928.$$

Target:

$$y = 1.75.$$

New loss:

$$\begin{aligned} & \frac{1}{2}(1.0928 - 1.75)^2 \\ &= 0.21595592. \end{aligned}$$

One small gradient step
reduced the loss
from
0.5
to about
0.216.

A Larger Step Can Overshoot

Use the same gradient
but choose
 $\alpha = 0.1$.

The updated parameters produce
 $\hat{y}_{\text{new}} = 4.205$.

The new loss is
3.0135125.

Thus:
a descent direction does not guarantee a decrease for an arbitrarily large step.

This is the same learning-rate issue
studied in Lesson 1.06.

Neural-network training inherits
all of those optimization concerns.

Learning Rate

The learning rate
 α
controls update size.

Too small:

training may be extremely slow.

Too large:

loss may oscillate;

updates may overshoot;

parameters may diverge.

A useful learning rate is often
among the most important training choices.

The optimal scale depends on:

network architecture;

feature scaling;

initialization;

batch size;

optimizer.

Learning-Rate Schedules

The learning rate need not remain constant.

A common strategy:
start larger
and reduce it during training.

Conceptually:

early training:
make substantial progress;

later training:
use smaller steps for refinement.

Possible schedules include:

piecewise reductions;

exponential decay;

inverse-time decay.

The specific schedule is a hyperparameter choice.

Validation performance should guide it.

Why Initialization Matters

A nonconvex network must begin
from some parameter vector
 θ_0 .

Different initializations can lead to:

different optimization trajectories;

different active ReLU patterns;

different final solutions.

Initialization is therefore not merely bookkeeping.

It affects whether useful gradients
can propagate through the network
at the beginning of training.

Do Not Initialize All Hidden Units Identically

Suppose two hidden units
begin with exactly the same incoming weights
and the same bias.

They produce identical activations.

Under the same downstream structure:
they receive identical gradients.

After one update:
they remain identical.

Thus the symmetry persists.

The two units behave like duplicated copies
instead of learning different features.

Random initialization breaks this symmetry.

Biases can often begin at zero;

weight matrices generally should not all begin identically.

Variance-Scaled Random Initialization

Weights should also not be arbitrarily large.

Large initial weights can create
large activations and gradients.

Tiny weights can make signals shrink
through deep networks.

Common heuristics scale variance
using the number of inputs
to the layer.

For ReLU-type activations:

a common He initialization uses approximately
$$\text{Var}(W_{ij}) \approx \frac{2}{n_{\text{in}}}.$$

For tanh-type networks:

Xavier / Glorot-style initialization
uses a scale on the order of
$$\frac{1}{n_{\text{in}}}$$

for weight variance,
with variants using both fan-in and fan-out.

These are practical initialization heuristics,
not guarantees of successful training.

A ReLU Can Become Inactive

For ReLU:
$$\sigma'(a) = 0$$

when
$$a < 0.$$

If a unit produces negative pre-activations
for essentially all training observations:
its incoming-weight gradient may vanish.

This is sometimes called a
dead ReLU.

Learning rate,
initialization,
and input scaling
can influence this behavior.

Variants such as leaky ReLU
retain a small negative-side slope,
but we will continue using ordinary ReLU
for our basic derivations.

Vanishing and Exploding Gradients

Backpropagation repeatedly multiplies by:

weight matrices
and

activation derivatives.

Across many layers:

products with norms consistently below 1
can shrink gradients dramatically;

products with large norms
can make gradients grow dramatically.

These are:
vanishing gradients
and
exploding gradients.

Initialization,
activations,
normalization methods,
and architecture
all affect these phenomena.

Gradient Clipping

If gradients occasionally become extremely large:

one practical safeguard is
gradient clipping.

For gradient vector

$\mathbf{g}$
and threshold

$C > 0$:

$$\tilde{\mathbf{g}} = \begin{cases} \mathbf{g}, & \|\mathbf{g}\| \leq C, \\ C \frac{\mathbf{g}}{\|\mathbf{g}\|}, & \|\mathbf{g}\| > C. \end{cases}.$$

This preserves gradient direction
while limiting its norm.

Clipping does not solve every instability,
but it can prevent a single extreme update
from destroying the parameter state.

Mini-Batch Size

Batch size

b
controls the same tradeoff
from Lesson 1.21.

Small batch:

cheap update;

noisy gradient;

many updates per epoch.

Large batch:

more stable gradient;

fewer updates per epoch;

more computation per update.

Modern hardware also matters:

matrix operations can make moderate mini-batches
computationally efficient.

Batch size is therefore both

a statistical

and computational hyperparameter.

Shuffle Each Epoch

If training observations arrive in a structured order:

all one class first;

sorted by response;

grouped by source;

then sequential mini-batches

may have systematically biased composition.

A common procedure:

shuffle the training indices at the beginning of each epoch.

Then divide the shuffled order

into mini-batches.

Validation and test data

are not mixed into this shuffle.

Regularization of Neural Networks

Neural networks can contain many parameters.

The same generalization concerns

from Lessons 1.27 and 1.29 remain.

A simple regularized objective:

$$g_{\text{reg}}(\theta) = g_{\text{data}}(\theta) + \frac{\lambda}{2} \sum_{\ell} \|W_{\ell}\|_F^2.$$

Usually:

weight matrices are penalized;

biases are often left unpenalized.

Here

$$\|W\|_F^2$$

is the sum of squares
of all entries of
 W .

Weight Decay Under Gradient Descent

For one weight matrix:

$$g_{\text{reg}} = g_{\text{data}} + \frac{\lambda}{2} \|W\|_F^2.$$

Then:

$$\nabla_W g_{\text{reg}} = \nabla_W g_{\text{data}} + \lambda W.$$

A gradient-descent step is

$$W_{\text{new}} = W - \alpha(\nabla_W g_{\text{data}} + \lambda W).$$

Rearrange:

$$W_{\text{new}} = (1 - \alpha\lambda)W - \alpha\nabla_W g_{\text{data}}.$$

Under ordinary gradient descent / SGD:

this multiplicative shrinkage
is the basic connection to
weight decay.

Training Loss Is Not Enough

As optimization proceeds:

training loss may continue decreasing.

But:

validation loss may eventually stop improving
and begin increasing.

That pattern suggests:

the model is fitting increasingly
sample-specific structure.

Therefore neural-network training should monitor:

training loss;

validation loss / metric;

epoch number;

sometimes gradient norms.

The validation set remains outside

parameter fitting.

Early Stopping

A common regularization strategy:
early stopping.

During training:

evaluate validation performance
after each epoch
or periodically.

Save the parameter state
with the best validation performance.

Stop when validation performance
has failed to improve
for a chosen number of checks.

That waiting period is often called
patience.

Then restore:
the parameters from the best validation epoch,

not merely the parameters
from the final epoch.

Early Stopping Is a Hyperparameter Procedure

If patience,
maximum epochs,
or stopping thresholds
are adjusted in response to validation performance:

they are part of model selection.

Therefore:
the validation set has influenced
the final fitted procedure.

The final test set must remain untouched
until those choices are complete.

This is the same separation
established in Lessons 1.29–1.30.

Momentum and Adaptive Optimizers

Plain mini-batch gradient descent is the foundation.

Practical neural-network training often uses
more elaborate update rules.

Examples:

momentum;

 RMSProp;

 Adam.

These methods use information
 from previous gradients
 and/or coordinatewise scaling
to modify the update.

We will not derive them here.

The important point:
 backpropagation computes the gradient; the optimizer decides how to use it.

A Small Network Training Example

Train a one-input ReLU network
 to approximate

$$y = |x|$$

on
 41
equally spaced points
 from
 −2
to
 2.

Use:

one hidden layer;

 4 ReLU units;

 linear regression output;

 half-MSE loss;

 full-batch gradient descent;

 learning rate 0.05;

 fixed random seed 0.

This is intentionally small
 so every training step can be written explicitly.

Training Code: Initialization

```
import numpy as np
```

```

def relu(a):
    return np.maximum(0, a)

def relu_prime(a):
    return (a > 0).astype(float)

x = np.linspace(-2, 2, 41)
X = x.reshape(-1, 1)

y = np.abs(x)

rng = np.random.default_rng(0)

m = 4
N = 1

W = rng.normal(
    0,
    np.sqrt(2/N),
    size=(m, N)
)

b = np.zeros(m)

v = rng.normal(
    0,
    np.sqrt(2/m),
    size=m
)

c = 0.0

```

The weight scales use
ReLU-oriented variance scaling.

Training Code: Forward and Backward Pass

```

def loss_and_gradients(
    X, y, W, b, v, c
):
    A = X @ W.T + b
    H = relu(A)

    y_hat = H @ v + c

    error = y_hat - y

    P = len(y)

    loss = 0.5 * np.mean(
        error**2
    )

    grad_v = H.T @ error / P
    grad_c = np.mean(error)

    Delta = (
        error[:, None]
        * v[None, :]
    ) * relu_prime(A)

```

```

grad_W = Delta.T @ X / P
grad_b = np.mean(
    Delta,
    axis=0
)

return (
    loss,
    grad_W,
    grad_b,
    grad_v,
    grad_c
)

```

Training Code: Gradient Updates

```

alpha = 0.05

for epoch in range(1000):

    (
        loss,
        grad_W,
        grad_b,
        grad_v,
        grad_c
    ) = loss_and_gradients(
        X, y, W, b, v, c
    )

    W -= alpha * grad_W
    b -= alpha * grad_b
    v -= alpha * grad_v
    c -= alpha * grad_c

```

Every epoch uses:

forward pass;

backpropagation;

gradient update.

The same structure scales
to deeper networks
and mini-batches.

Observed Training Loss

For this fixed initialization:

Epoch	Half-MSE Training Loss
0	0.32354
10	0.19974
50	0.04718
100	0.01917
500	0.000119
999	0.0000273

The network learns
 a near-exact representation
 of
 $|x|$.

This is consistent with Lesson 1.32:
 two appropriately arranged ReLU units
 are already sufficient
 to represent
 $|x|$
 exactly.

Training discovers
 a useful parameterization
 numerically.

Convert the Loop to Mini-Batches

For larger datasets:

```

rng = np.random.default_rng(0)

batch_size = 32

for epoch in range(num_epochs):

    order = rng.permutation(len(X))

    for start in range(
        0,
        len(X),
        batch_size
    ):
        idx = order[
            start:start+batch_size
        ]

        Xb = X[idx]
        yb = y[idx]

        # forward pass
        # backpropagation
        # update parameters

```

This is the same shuffled mini-batch structure
 from Lesson 1.21,
 now combined with backpropagation.

A Practical Training Checklist

Before training:

- scale inputs appropriately;
- choose architecture and activation;
- initialize weights sensibly;
- reserve validation data correctly.

During training:

- shuffle;
- form mini-batches;
- forward;
- backpropagate;
- update;
- monitor losses and metrics.

After training:

- restore the best validation checkpoint if using early stopping;
- evaluate once on untouched test data.

What did we learn?

- ◆ Neural-network training repeatedly combines a forward pass, loss evaluation, backpropagation, and a parameter update.
- ◆ Mini-batch training applies the finite-sum optimization ideas from stochastic gradient methods to gradients produced by backpropagation.
- ◆ Learning rate controls step size: a small step can reduce loss while a larger step in the same gradient direction can overshoot.
- ◆ Random variance-scaled initialization breaks hidden-unit symmetry and helps keep activations and gradients at useful numerical scales.
- ◆ Weight penalties, validation monitoring, and early stopping help control generalization rather than merely training fit.
- ◆ Backpropagation supplies gradients; the optimizer, batch structure, learning-rate schedule, and stopping rule determine how those gradients are used.



Next

- Neural networks build nonlinear predictors by learning internal feature representations.
- The final major supervised-learning family in this course takes a very different approach: recursively partition

the input space.

- Decision trees make predictions using simple if–then splits rather than smooth feature compositions.
- We next develop the geometry and fitting rule of a decision tree.
- **1.35 - Decision Trees**