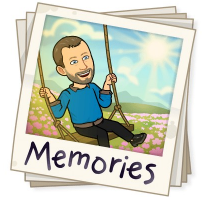


Machine Learning

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

Previous Lecture

- ♦ A neural network learns a feature map by composing trainable affine transformations with nonlinear activation functions.
- ♦ A dense layer has the form $\mathbf{h} = \sigma(W\mathbf{x} + \mathbf{b})$, with one row of W for each hidden unit.
- ♦ Without nonlinear activations, stacked affine layers collapse to a single affine transformation and gain no expressive power.
- ♦ Width counts units in a layer; depth counts successive hidden transformations under our stated convention; layer sizes determine the number of trainable parameters.
- ♦ Regression, binary classification, and multiclass classification use familiar output heads applied to the learned final hidden representation.
- ♦ The forward pass evaluates the network; training requires differentiating the loss through all layers, which leads to the chain rule and backpropagation.



§13.4 - The Chain Rule and Backpropagation

Last time:

a neural network computed a forward pass

$$\mathbf{x} \rightarrow \mathbf{a}_1 \rightarrow \mathbf{h}_1 \rightarrow \cdots \rightarrow \hat{y}.$$

Training requires derivatives

with respect to every weight and bias.

A large network may contain millions of parameters.

We do not want to differentiate the full expression separately
for every parameter.

Today:

use the chain rule

and reuse intermediate derivatives.

This is **backpropagation**.

The Scalar Chain Rule

Suppose

$$z = f(y)$$

and

$$y = g(x).$$

Then

$$\frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx}.$$

A neural network is simply a large composition of functions.

Backpropagation applies this rule repeatedly,
moving from the loss backward through the composition.

A One-Hidden-Unit Network

Let

$$\begin{aligned}a &= wx + b, \\h &= \sigma(a), \\\hat{y} &= vh + c,\end{aligned}$$

and for one regression observation use

$$L = \frac{1}{2}(\hat{y} - y)^2.$$

The factor 1/2 simplifies derivatives.

The dependency path for w is

$$w \rightarrow a \rightarrow h \rightarrow \hat{y} \rightarrow L.$$

Therefore

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h} \frac{\partial h}{\partial a} \frac{\partial a}{\partial w}.$$

Local Derivatives

For

$$\begin{aligned}L &= \frac{1}{2}(\hat{y} - y)^2: \\\frac{\partial L}{\partial \hat{y}} &= \hat{y} - y.\end{aligned}$$

For

$$\begin{aligned}\hat{y} &= vh + c: \\\frac{\partial \hat{y}}{\partial v} &= h, \quad \frac{\partial \hat{y}}{\partial h} = v, \quad \frac{\partial \hat{y}}{\partial c} = 1.\end{aligned}$$

For

$$\begin{aligned}h &= \sigma(a): \\\frac{\partial h}{\partial a} &= \sigma'(a).\end{aligned}$$

For

$$\begin{aligned}a &= wx + b: \\\frac{\partial a}{\partial w} &= x, \quad \frac{\partial a}{\partial b} = 1.\end{aligned}$$

Scalar Backpropagation

Define the output error

$$\delta_{\text{out}} = \frac{\partial L}{\partial \hat{y}} = \hat{y} - y.$$

Then

$$\begin{aligned}\frac{\partial L}{\partial v} &= \delta_{\text{out}} h, \\\frac{\partial L}{\partial c} &= \delta_{\text{out}}.\end{aligned}$$

Backpropagate through the hidden unit:

$$\delta = \frac{\partial L}{\partial a} = \delta_{\text{out}} v \sigma'(a).$$

Finally:

$$\frac{\partial L}{\partial w} = \delta x, \quad \frac{\partial L}{\partial b} = \delta.$$

The intermediate quantity δ is reused.

ReLU Derivative

For

$$\text{ReLU}(a) = \max(0, a):$$

$$\text{ReLU}'(a) = \begin{cases} 0, & a < 0, \\ 1, & a > 0. \end{cases}.$$

At $a = 0$ the derivative does not exist.

Implementations choose a convention,
commonly 0.

For hand calculations we will avoid the kink
unless the convention is stated explicitly.

Reuse the Network From Lesson 1.32

Take

$$\mathbf{x} = \begin{bmatrix} 2 \\ 1 \end{bmatrix},$$

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

$$\mathbf{b} = \begin{bmatrix} 0 \\ -1/2 \end{bmatrix},$$

$$\mathbf{v} = \begin{bmatrix} 2 \\ -1 \end{bmatrix},$$

$$c = 1/4.$$

Lesson 1.32 gave

$$\mathbf{a} = (1, 3/2)^T,$$

$$\mathbf{h} = (1, 3/2)^T,$$

$$\hat{y} = 3/4.$$

Now choose target

$$y = 7/4.$$

Then

$$\delta_{\text{out}} = \hat{y} - y = -1,$$

and

$$L = \frac{1}{2}.$$

Output-Layer Gradients for the Example

Because

$$\hat{y} = \mathbf{v}^T \mathbf{h} + c:$$
$$\nabla_{\mathbf{v}} L = \delta_{\text{out}} \mathbf{h} = \begin{bmatrix} -1 \\ -3/2 \end{bmatrix}.$$

Also

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

The derivative with respect to the hidden activation is

$$\nabla_{\mathbf{h}} L = \delta_{\text{out}} \mathbf{v} = \begin{bmatrix} -2 \\ 1 \end{bmatrix}.$$

The Hadamard Product and Hidden Error

The symbol

$$\odot$$

means componentwise multiplication.

For vectors

$$\mathbf{r}, \mathbf{s} \in \mathbb{R}^m:$$
$$\mathbf{r} \odot \mathbf{s} = (r_1 s_1, \dots, r_m s_m)^T.$$

Because both pre-activations in our example are positive:

$$\sigma'(\mathbf{a}) = (1, 1)^T.$$

Thus

$$\boldsymbol{\delta} = \nabla_{\mathbf{a}} L = (\delta_{\text{out}} \mathbf{v}) \odot \sigma'(\mathbf{a}) = \begin{bmatrix} -2 \\ 1 \end{bmatrix}.$$

Weight-Matrix and Bias Gradients

For

$$\mathbf{a} = W\mathbf{x} + \mathbf{b}:$$
$$a_i = \sum_j W_{ij} x_j + b_i.$$

Hence

$$\frac{\partial L}{\partial W_{ij}} = \delta_i x_j.$$

Collect all entries:

$$\nabla_W L = \boldsymbol{\delta} \mathbf{x}^T.$$

For the example:

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

And

$$\nabla_{\mathbf{b}} L = \boldsymbol{\delta} = \begin{bmatrix} -2 \\ 1 \end{bmatrix}.$$

One-Hidden-Layer Backpropagation Formula

For

$$\begin{aligned} \mathbf{a} &= W\mathbf{x} + \mathbf{b}, \\ \mathbf{h} &= \sigma(\mathbf{a}), \\ \hat{y} &= \mathbf{v}^T \mathbf{h} + c, \\ L &= \frac{1}{2}(\hat{y} - y)^2: \end{aligned}$$

$$\delta_{\text{out}} = \hat{y} - y,$$

$$\nabla_{\mathbf{v}} L = \delta_{\text{out}} \mathbf{h},$$

$$\frac{\partial L}{\partial c} = \delta_{\text{out}},$$

$$\boldsymbol{\delta} = (\delta_{\text{out}} \mathbf{v}) \odot \sigma'(\mathbf{a}),$$

$$\nabla_W L = \boldsymbol{\delta} \mathbf{x}^T,$$

$$\nabla_{\mathbf{b}} L = \boldsymbol{\delta}.$$

The outer product has exactly the same dimensions as W .

Two Hidden Layers

Let

$$\begin{aligned} \mathbf{a}_1 &= W_1 \mathbf{x} + \mathbf{b}_1, \\ \mathbf{h}_1 &= \sigma_1(\mathbf{a}_1), \\ \mathbf{a}_2 &= W_2 \mathbf{h}_1 + \mathbf{b}_2, \\ \mathbf{h}_2 &= \sigma_2(\mathbf{a}_2), \\ \hat{y} &= \mathbf{v}^T \mathbf{h}_2 + c. \end{aligned}$$

Start at the output:

$$\delta_{\text{out}} = \hat{y} - y.$$

Second hidden layer:

$$\begin{aligned} \boldsymbol{\delta}_2 &= (\delta_{\text{out}} \mathbf{v}) \odot \sigma_2'(\mathbf{a}_2), \\ \nabla_{W_2} L &= \boldsymbol{\delta}_2 \mathbf{h}_1^T, \\ \nabla_{\mathbf{b}_2} L &= \boldsymbol{\delta}_2. \end{aligned}$$

Backpropagate to the First Hidden Layer

Since

$$\begin{aligned} \mathbf{a}_2 &= W_2 \mathbf{h}_1 + \mathbf{b}_2: \\ \nabla_{\mathbf{h}_1} L &= W_2^T \boldsymbol{\delta}_2. \end{aligned}$$

Pass through the first activation:

$$\boldsymbol{\delta}_1 = (W_2^T \boldsymbol{\delta}_2) \odot \sigma_1'(\mathbf{a}_1).$$

Then

$$\nabla_{W_1} L = \boldsymbol{\delta}_1 \mathbf{x}^T,$$

$$\nabla_{\mathbf{b}_1} L = \boldsymbol{\delta}_1.$$

The same two operations repeat:

transpose-weight propagation

followed by activation differentiation.

General Backpropagation Recursion

Let

$$\mathbf{h}_0 = \mathbf{x},$$

and for hidden layers

$$\ell = 1, \dots, L:$$

$$\mathbf{a}_\ell = W_\ell \mathbf{h}_{\ell-1} + \mathbf{b}_\ell,$$

$$\mathbf{h}_\ell = \sigma_\ell(\mathbf{a}_\ell).$$

Once

$$\boldsymbol{\delta}_{\ell+1} = \nabla_{\mathbf{a}_{\ell+1}} L$$

is known:

$$\boldsymbol{\delta}_\ell = (W_{\ell+1}^T \boldsymbol{\delta}_{\ell+1}) \odot \sigma'_\ell(\mathbf{a}_\ell).$$

Then

$$\nabla_{W_\ell} L = \boldsymbol{\delta}_\ell \mathbf{h}_{\ell-1}^T,$$

$$\nabla_{\mathbf{b}_\ell} L = \boldsymbol{\delta}_\ell.$$

Why the Forward Pass Stores Intermediate Values

Backpropagation needs

$$\mathbf{h}_{\ell-1}$$

and

$$\sigma'_\ell(\mathbf{a}_\ell).$$

Therefore the forward pass stores:

pre-activations

$$\mathbf{a}_\ell;$$

activations

$$\mathbf{h}_\ell.$$

The backward pass reuses them.

This shared computation is why backpropagation is efficient.

It is reverse-mode automatic differentiation

specialized to a layered computational graph.

Binary Classification Output Error

For a neural classifier:

$$f = \mathbf{v}^T \mathbf{h} + c,$$

$$p = \sigma(f),$$

with binary cross-entropy.

From Lesson 1.16:

$$\frac{\partial L}{\partial f} = p - y.$$

Thus:

$$\delta_{\text{out}} = p - y.$$

Then

$$\nabla_{\mathbf{v}} L = (p - y) \mathbf{h},$$

$$\partial L / \partial c = p - y,$$

and the hidden-layer recursion is unchanged.

Softmax Cross-Entropy Output Error

For multiclass classification:

$$\mathbf{f} \in \mathbb{R}^K,$$

$$\mathbf{p} = \text{softmax}(\mathbf{f}),$$

$\mathbf{y}$ one-hot.

From Lesson 1.20:

$$\nabla_{\mathbf{f}} L = \mathbf{p} - \mathbf{y}.$$

So:

$$\delta_{\text{out}} = \mathbf{p} - \mathbf{y}.$$

Regression,

binary classification,

and multiclass classification

differ mainly in how the output delta is initialized.

Batch Backpropagation

For row-oriented data:

$$H_{\ell-1} \in \mathbb{R}^{P \times n_{\ell-1}},$$

$$W_{\ell} \in \mathbb{R}^{n_{\ell} \times n_{\ell-1}},$$

and

$$\Delta_{\ell} \in \mathbb{R}^{P \times n_{\ell}}$$

contains one pre-activation error row per observation.

For a mean loss:

$$\nabla_{W_{\ell}} g = \frac{1}{P} \Delta_{\ell}^T H_{\ell-1},$$

$$\nabla_{\mathbf{b}_{\ell}} g = \frac{1}{P} \Delta_{\ell}^T \mathbf{1}.$$

Backward recursion:

$$\Delta_{\ell-1} = (\Delta_{\ell} W_{\ell}) \odot \sigma'_{\ell-1}(A_{\ell-1}).$$

Gradient Checking

For parameter

$$\theta_j,$$

centered finite differences give

$$\frac{\partial L}{\partial \theta_j} \approx \frac{L(\theta_j + \epsilon) - L(\theta_j - \epsilon)}{2\epsilon}.$$

Use this on a small network
to verify analytic backpropagation.

But:
for D parameters,
centered finite differences require roughly
 $2D$
extra forward evaluations.

Thus finite differences are for checking,
not for training.

Backpropagation in Python

```
import numpy as np

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

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

x = np.array([2., 1.])

W = np.array([[1., -1.],
               [0.5, 1.]])

b = np.array([0., -0.5])

v = np.array([2., -1.])

c = 0.25
y = 1.75

a = W @ x + b
h = relu(a)
y_hat = v @ h + c

delta_out = y_hat - y

grad_v = delta_out * h
grad_c = delta_out

delta = (
    delta_out * v
) * relu_prime(a)

grad_W = np.outer(delta, x)
grad_b = delta
```

Results:

$$\nabla_{\mathbf{v}} L = (-1, -1.5)^T,$$

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

$$\nabla_{\mathbf{b}} L = (-2, 1)^T,$$

$$\partial L / \partial c = -1.$$

New Python Idea: Outer Product

```
np.outer(delta, x)
```

constructs

$$\delta \mathbf{x}^T.$$

For

$$\delta = (-2, 1)^T$$

and

$$\mathbf{x} = (2, 1)^T: \\ \begin{bmatrix} -4. & -2. \\ 2. & 1. \end{bmatrix}$$

The result has the same dimensions as

$$W.$$

Finite-Difference Check in Python

```
def loss_value(W, b, v, c, x, y):  
    a = W @ x + b  
    h = relu(a)  
    y_hat = v @ h + c  
  
    return 0.5 * (y_hat - y)**2
```

```
eps = 1e-6
```

```
W_plus = W.copy()  
W_minus = W.copy()
```

```
W_plus[0, 0] += eps  
W_minus[0, 0] -= eps
```

```
numeric = (  
    loss_value(W_plus, b, v, c, x, y)  
    -  
    loss_value(W_minus, b, v, c, x, y)  
) / (2*eps)
```

```
print(numeric)  
print(grad_W[0, 0])
```

Both values are approximately

$$-4.$$

A full gradient check repeats this
for every parameter.

What Backpropagation Does Not Do

Backpropagation computes

$$\nabla_{\theta} L.$$

It does not decide:

the learning rate;

full-batch versus mini-batch training;

parameter initialization;

regularization;

stopping criteria.

Those are training and optimization decisions.

Backpropagation supplies the gradients
those methods require.

What did we learn?

- ◆ 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 $\nabla_{b_\ell} L = \delta_\ell$.
- ◆ 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.



Next

- Backpropagation gives the gradient of a neural-network loss with respect to every weight and bias.
- We now need to turn those gradients into a practical training procedure.
- Initialization, mini-batches, learning rates, regularization, and stopping criteria all affect whether optimization succeeds.
- Next we combine the optimization tools from earlier lessons with neural-network gradients.
- **1.34 - Training Neural Networks**