



CONTENTS

1	Neural Networks for Regression	3
1.1	From Linear Models to Networks	3
1.2	Activations	4
1.3	Training	5
1.4	PyTorch: Tensors and Automatic Gradients	5
1.5	Learning a Day of Temperatures	5
1.6	Predicting Penguin Mass	8
2	Neural Networks for Classification	13
2.1	The Output Layer	13
2.2	Cross-Entropy Loss	14
2.3	Classifying Penguins with PyTorch	14
2.4	Bending the Boundary	16
3	Deep Learning	19
3.1	Many Layers	19
3.2	The Chain Rule	19
3.3	Backpropagation	20
3.4	Vanishing Gradients	21
3.5	Convolutions: Networks That See	22
3.6	A Field Guide to Architectures	24
4	Statistical Tables	29
4.1	The Standard Normal Table	29

4.2	The t Table	32
4.3	The Chi-Square Table	34
A	Answers to Exercises	37
Index		41

Neural Networks for Regression

Plot the temperature over a day: it bottoms out before dawn, climbs through the morning, peaks in the afternoon, and falls at night. No straight line fits that. You could guess a formula, maybe a cosine, but what if you have no idea what shape to expect? Neural networks build curves out of many simple pieces, and they learn the pieces from data.

Neural networks are behind image recognition, speech recognition, and chatbots. Underneath, they use ideas you already know from this unit: a linear model, a loss function, and gradient descent. This chapter puts those pieces together and trains networks with *PyTorch*, the library many researchers use.

1.1 From Linear Models to Networks

A *neuron* computes a weighted sum of its inputs plus a bias, then applies an *activation function* σ :

$$a = \sigma(w_1x_1 + w_2x_2 + \cdots + w_px_p + b)$$

With the identity activation, a single neuron is just linear regression. With the sigmoid, it is logistic regression. The new idea is to *stack* neurons in layers, so the outputs of one layer are the inputs to the next.

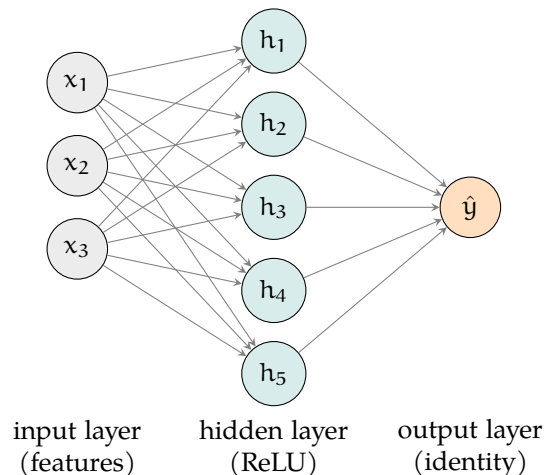


Figure 1.1: A network with 3 inputs, one hidden layer of 5 neurons, and 1 output. Each arrow carries a weight.

- The **input layer** holds the features. It does no computing.

- Each **hidden layer** computes new features from the previous layer. In matrix form, a layer is $\mathbf{h} = \sigma(W\mathbf{x} + \mathbf{b})$, with one row of W for each neuron.
- The **output layer** produces the prediction. For regression, it uses the identity activation, because the prediction can be any real number.

The network in Figure 1.1 has $3 \times 5 + 5 = 20$ parameters in the hidden layer (weights plus biases) and $5 + 1 = 6$ in the output layer: 26 numbers to learn.

1.2 Activations

Without a nonlinear activation, stacking layers gains nothing: a linear function of a linear function is still linear, so the whole network would collapse into one linear regression. The activation is what lets a network bend.

Name	Formula	Range
ReLU (rectified linear unit)	$\max(0, z)$	$[0, \infty)$
sigmoid	$1/(1 + e^{-z})$	$(0, 1)$
hyperbolic tangent	$\tanh z$	$(-1, 1)$
identity (output, for regression)	z	all real numbers

The ReLU is the most common choice for hidden layers today. It is cheap to compute, and its slope is exactly 1 for positive inputs, which keeps gradients from shrinking as they pass back through many layers.

Example: Here is a tiny network with one input x , two hidden ReLU neurons, and one output:

$$h_1 = \max(0, x - 1), \quad h_2 = \max(0, 1 - x), \quad \hat{y} = h_1 + h_2$$

What function does it compute?

Solution: Make a table.

x	h_1	h_2	$\hat{y}$
-1	0	2	2
0	0	1	1
1	0	0	0
2	1	0	1
3	2	0	2

The output is $\hat{y} = |x - 1|$, a V shape. Each hidden neuron contributes one straight piece, and the ReLU switches it on only on its side of $x = 1$. No single line can make a V, but two neurons can. With more neurons, a network of ReLUs can bend in more places, and it can approximate almost any continuous curve as closely as you like.

1.3 Training

Training a network follows the same steps as any gradient-based model:

1. **Feedforward:** compute the predictions from the inputs, layer by layer.
2. **Loss:** compare predictions with targets, usually with mean squared error for regression.
3. **Backpropagation:** compute the gradient of the loss with respect to every weight. (The Deep Learning chapter shows how the chain rule makes this efficient.)
4. **Update:** move each weight a small step against its gradient.

Unlike least squares, the loss is no longer a simple bowl; it can have many valleys. So the result depends on the random starting weights, and there is no formula for the answer. Gradient descent is the only way down.

1.4 PyTorch: Tensors and Automatic Gradients

Computing gradients by hand for thousands of weights would be miserable. PyTorch does it for you. Its basic object is the *tensor*, an array very much like a NumPy array. If you create a tensor with `requires_grad=True`, PyTorch records every calculation done with it, and `backward()` works out the derivatives. This is called *automatic differentiation*. Here is `autograd_demo.py`:

```

1 import torch
2
3 x = torch.tensor(3.0, requires_grad=True)
4 y = x**2 + 2 * x           # PyTorch records how y was computed
5 y.backward()               # ...so it can work out dy/dx
6 print(y.item(), x.grad.item())

```

15.0 8.0

At $x = 3$, $y = 9 + 6 = 15$ and $\frac{dy}{dx} = 2x + 2 = 8$. PyTorch got the derivative without being told the formula. (Install PyTorch with `pip install torch`.)

1.5 Learning a Day of Temperatures

The file `temperature.py` fits a network with 16 hidden neurons to a day of temperature readings, one every 15 minutes. (The data is simulated: a cosine curve plus random noise with a standard deviation of 0.8 °C.) The network is never told about cosines.

```

1 import numpy as np
2 import torch
3 from torch import nn
4
5 # Simulated data: temperature (degrees C) every 15 minutes for one day, with noise
6 rng = np.random.default_rng(0)
7 hours = np.arange(0, 24, 0.25)
8 temps = 12 - 6 * np.cos(2 * np.pi * (hours - 3) / 24) + rng.normal(0, 0.8,
9 ↪ len(hours))
10
11 # Scale the input to -1..1 and standardize the target
12 mean, sd = temps.mean(), temps.std()
13 x = torch.tensor((hours - 12) / 12, dtype=torch.float32).unsqueeze(1) # shape
14 ↪ (96, 1)
15 y = torch.tensor((temps - mean) / sd, dtype=torch.float32).unsqueeze(1)
16
17 torch.manual_seed(0)
18 model = nn.Sequential(
19     nn.Linear(1, 16),      # 1 input -> 16 hidden neurons
20     nn.ReLU(),
21     nn.Linear(16, 1),      # 16 hidden neurons -> 1 output
22 )
23 optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
24 loss_fn = nn.MSELoss()
25
26 for epoch in range(1, 2001):
27     optimizer.zero_grad()      # clear old gradients
28     loss = loss_fn(model(x), y) # feedforward and loss
29     loss.backward()            # backpropagation
30     optimizer.step()           # update the weights
31     if epoch in (1, 10, 100, 2000):
32         print(f"epoch {epoch:4d}: RMSE {loss.item() ** 0.5 * sd:.2f} degrees")
33
34 with torch.no_grad():
35     for h in [3, 15]:
36         guess = model(torch.tensor([(h - 12) / 12])).item() * sd + mean
37         print(f"{h:2d}:00 predicted {guess:.1f} degrees")

```

```

epoch    1: RMSE 4.16 degrees
epoch   10: RMSE 3.60 degrees
epoch  100: RMSE 1.79 degrees
epoch 2000: RMSE 0.75 degrees
   3:00 predicted 5.4 degrees
  15:00 predicted 17.8 degrees

```

(PyTorch's random numbers can differ between versions and computers, so your numbers may differ a little.)

The pieces:

- A linear layer computes $W\mathbf{x}+\mathbf{b}$; `nn.Linear(1, 16)` has a 16×1 weight matrix. `nn.Sequential` chains layers into a network. `model(x)` runs the feedforward step on all 96 readings at once.
- `unsqueeze(1)` turns a list of 96 numbers into a 96×1 column, because PyTorch expects one row per example and one column per feature.
- Adam is an improved version of gradient descent that adapts the step size for each weight. It is the usual default.
- The four lines in the loop are the four training steps. `zero_grad` is needed because PyTorch *adds* new gradients to old ones unless you clear them.
- `torch.no_grad()` tells PyTorch not to record calculations when we only want predictions.

The final RMSE, 0.75°C , is about the size of the noise we added, so the network has found the underlying curve. The true temperatures at 3:00 and 15:00 are 6 and 18 degrees.

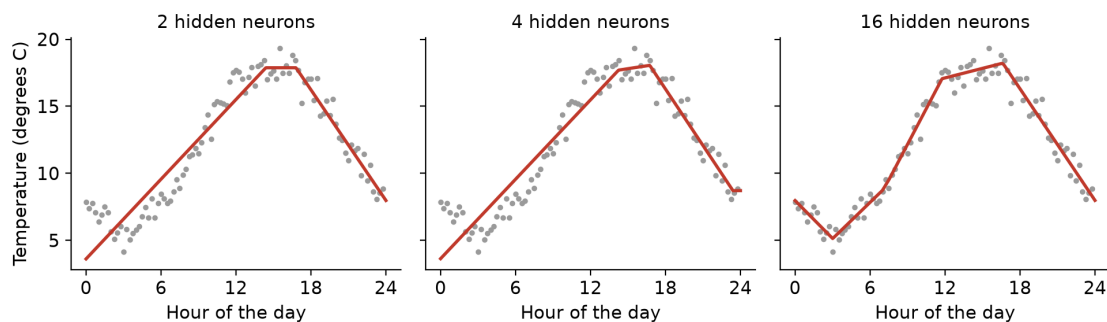


Figure 1.2: Networks with 2, 4, and 16 hidden ReLU neurons fit to the simulated temperatures. Each is the best of five random starts.

Figure 1.2 shows how the width matters. Every network of ReLUs draws a broken line. Two neurons can make only a couple of bends; sixteen can follow the dip before dawn. Why “best of five starts”? With some starting weights, a ReLU neuron’s input is negative for every data point, so its output and its gradient are always zero, and it never learns anything: a *dead neuron*. Trying several starts, or using more neurons than you strictly need, protects against that.

Notice the scaling: the input is squeezed into -1 to 1 , and the target is standardized. As in the Gradient Descent chapter, gradient-based training is much happier when the numbers are near 1 in size. Forget to scale the target, and this network gets stuck near an RMSE of 3.6 degrees.

1.6 Predicting Penguin Mass

Now a network for penguin body mass, using the same six features as the multiple regression and boosting examples: flipper length, bill length, bill depth, sex, and two species columns. Networks with many weights overfit easily, so we hold out a validation set and use *early stopping*: train for a long time, but keep the weights from the epoch with the best validation error. Here is `penguin_net.py`:

```

1  import copy
2  import numpy as np
3  import torch
4  from torch import nn
5  from penguins import load_penguins
6
7  data = load_penguins()
8  columns = [data["flipper_length_mm"], data["bill_length_mm"],
9  ↪ data["bill_depth_mm"],
10             data["sex"] == "male", data["species"] == "Gentoo", data["species"] ==
11             ↪ "Chinstrap"]
12  X = np.column_stack(columns).astype(float)
13  y = data["body_mass_g"]
14
15  order = np.random.default_rng(1).permutation(len(y))
16  train, val, test = order[:183], order[183:233], order[233:]
17
18  # Standardize with training statistics only
19  x_mean, x_sd = X[train].mean(axis=0), X[train].std(axis=0)
20  y_mean, y_sd = y[train].mean(), y[train].std()
21
22  def tensors(rows):
23      xs = torch.tensor((X[rows] - x_mean) / x_sd, dtype=torch.float32)
24      ys = torch.tensor((y[rows] - y_mean) / y_sd, dtype=torch.float32).unsqueeze(1)
25      return xs, ys
26
27  (x_tr, y_tr), (x_va, y_va), (x_te, y_te) = tensors(train), tensors(val),
28  ↪ tensors(test)
29
30  torch.manual_seed(0)
31  model = nn.Sequential(nn.Linear(6, 32), nn.ReLU(), nn.Linear(32, 32), nn.ReLU(),
32                        nn.Linear(32, 1))
33  optimizer = torch.optim.Adam(model.parameters(), lr=0.003)
34  loss_fn = nn.MSELoss()
35
36  def rmse_grams(x, y):
37      with torch.no_grad():
38          return loss_fn(model(x), y).item() ** 0.5 * y_sd
39
40  best_val, best_state, best_epoch = np.inf, None, 0
41  for epoch in range(1, 1001):
42      optimizer.zero_grad()
43      loss = loss_fn(model(x_tr), y_tr)
44      loss.backward()
45      optimizer.step()
46      val_rmse = rmse_grams(x_va, y_va)

```

```

41     if val_rmse < best_val:                # remember the best weights so far
42         best_val, best_epoch = val_rmse, epoch
43         best_state = copy.deepcopy(model.state_dict())
44     if epoch in (10, 100, 300, 1000):
45         print(f"epoch {epoch:4d}: training {rmse_grams(x_tr, y_tr):4.0f} g,"
46               f" validation {val_rmse:4.0f} g")
47
48 model.load_state_dict(best_state)          # early stopping
49 print(f"best validation RMSE {best_val:.0f} g at epoch {best_epoch}")
50 print(f"test RMSE {rmse_grams(x_te, y_te):.0f} g")
51 print("parameters:", sum(p.numel() for p in model.parameters()))

```

```

epoch 10: training 599 g, validation 579 g
epoch 100: training 262 g, validation 309 g
epoch 300: training 237 g, validation 327 g
epoch 1000: training 140 g, validation 454 g
best validation RMSE 307 g at epoch 56
test RMSE 262 g
parameters: 1313

```

This network has two hidden layers of 32 neurons and 1313 parameters, trained on just 183 penguins. Left alone, it overfits badly: by epoch 1000, it is off by only 140 g on the training penguins but 454 g on the validation penguins. A `state_dict` holds all of a model's weights, so saving a copy whenever validation improves, and loading the best one at the end, gives us the model from epoch 56.

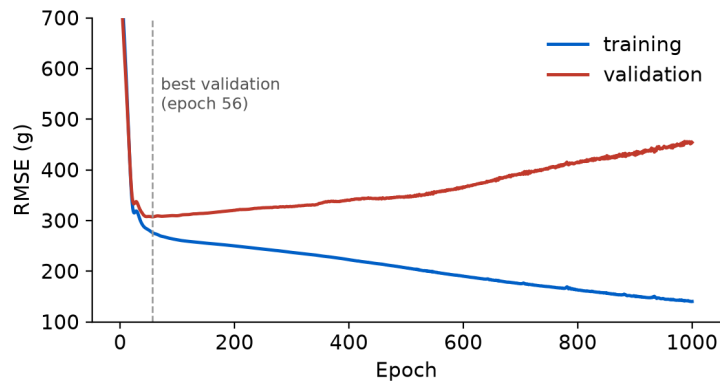


Figure 1.3: Training and validation RMSE for the penguin network.

Model (trained on the same 183 penguins)	Test RMSE (g)
always guess the average	about 800
least squares, same six features	282
neural network, early stopping	262

The network edges out least squares, by picking up some curvature and interactions the

straight-line model cannot. But the gain is small, and the network took more care: scaling, a validation set, early stopping, and a random seed. With a few hundred rows of tabular data, a neural network is rarely the clear winner. Networks shine when there is a lot of data with complicated structure, like images, sound, and text.

Exercise 1 Compute the error

A neuron predicts $\hat{y} = 2x + 1$. For the point $(2, 4)$, what is the squared error?

Working Space

Answer on Page 37

Exercise 2 Output activation

Why is the identity activation used in the output layer for regression, but not the sigmoid?

Working Space

Answer on Page 37

Exercise 3 Build a tent

A network has two ReLU neurons, $h_1 = \max(0, x)$ and $h_2 = \max(0, x - 2)$, and the output $\hat{y} = h_1 - 2h_2$. Compute $\hat{y}$ at $x = 0, 1, 2, 3, 4$. Describe the shape.

Working Space

Answer on Page 37

Exercise 4 **Count the parameters**

A network has 6 inputs, a hidden layer of 32 neurons, a second hidden layer of 32 neurons, and 1 output. How many weights and biases does it have? Check your answer against the output of `penguin_net.py`.

*Working Space**Answer on Page 37***Exercise 5** **Collapse**

Show that two layers with no activation, $\mathbf{h} = \mathbf{W}_1\mathbf{x} + \mathbf{b}_1$ and $\hat{\mathbf{y}} = \mathbf{W}_2\mathbf{h} + \mathbf{b}_2$, are equivalent to a single linear layer.

*Working Space**Answer on Page 37***Exercise 6** **Try other starts**

In `temperature.py`, change the random seed from 0 to each of 1 through 5. Do all of them reach an RMSE near 0.75 degrees? For any that do not, what might have happened?

*Working Space**Answer on Page 38*

Neural Networks for Classification

When your phone unlocks for your face but not your sibling's, or a photo app sorts pictures into "beach," "food," and "pets," a neural network is classifying. The network looks just like the regression networks of the last chapter. Only two things change: the output layer, which now produces probabilities, and the loss, which now grades probabilities.

2.1 The Output Layer

For two classes, a single output neuron uses the sigmoid activation, $\sigma(z) = \frac{1}{1+e^{-z}}$, which lies between 0 and 1. That is logistic regression, with hidden layers in front of it computing better features.

For K classes, the output layer has one neuron per class. Their raw outputs $z_1, \dots, z_K$, called *logits*, can be any real numbers. The *softmax* function turns them into probabilities that are positive and sum to 1:

$$p_k = \frac{e^{z_k}}{e^{z_1} + e^{z_2} + \dots + e^{z_K}}$$

Exponentiating makes everything positive, and dividing by the total makes the probabilities add up to 1. The biggest logit always gets the biggest probability.

Example: A network that sorts photos into cat, dog, or bird produces logits $\mathbf{z} = (2, 1, 0)$. Find the probabilities.

Solution: Exponentiate each score and divide by the total:

$$e^2 \approx 7.389, \quad e^1 \approx 2.718, \quad e^0 = 1, \quad \text{total} \approx 11.107$$

$$P(\text{cat}) \approx \frac{7.389}{11.107} \approx 0.665, \quad P(\text{dog}) \approx 0.245, \quad P(\text{bird}) \approx 0.090$$

Adding the same number to every logit does not change the probabilities (it multiplies the top and bottom by the same factor), so only the differences between logits matter. With two classes, softmax reduces to the sigmoid of the difference $z_1 - z_2$.

2.2 Cross-Entropy Loss

The loss is the same log loss you met in the Logistic Regression chapter, extended to K classes: the negative log of the probability the network gave to the *correct* class.

$$L = -\ln p_{\text{true class}}$$

For a binary label $y \in \{0, 1\}$ and predicted probability p , this is the familiar

$$L = -[y \ln p + (1 - y) \ln(1 - p)]$$

If the photo in the example really is a cat, the loss is $-\ln 0.665 \approx 0.408$. If it is a bird, the loss is $-\ln 0.090 \approx 2.41$: confidently wrong costs a lot.

Probability given to the correct class	Cross-entropy loss
0.99	0.010
0.9	0.105
0.5	0.693
0.1	2.303
0.01	4.605

Example: A single neuron computes $z = wx + b$ with $w = 2$, $b = -1$, and input $x = 1$. Find the predicted probability of class 1 and the loss if the true label is 1.

Solution: $z = 1$, and

$$p = \sigma(1) = \frac{1}{1 + e^{-1}} \approx 0.731$$

The cross-entropy loss is $-\ln(0.731) \approx 0.313$.

2.3 Classifying Penguins with PyTorch

Here is `species_net.py`, which predicts species from the four measurements. It uses one hidden layer of 16 ReLUs and three outputs, and trains with mini-batches of 32 penguins, as in the Linear Regression and Gradient Descent chapter:

```

1 import numpy as np
2 import torch
3 from torch import nn
4 from penguins import load_penguins
5
6 data = load_penguins()
7 names, labels = np.unique(data["species"], return_inverse=True)
8 X = np.column_stack([data["bill_length_mm"], data["bill_depth_mm"],
9                      data["flipper_length_mm"], data["body_mass_g"]])
10 order = np.random.default_rng(1).permutation(len(labels))

```

```

11 train, test = order[:233], order[233:]
12
13 mean, sd = X[train].mean(axis=0), X[train].std(axis=0)
14 x_tr = torch.tensor((X[train] - mean) / sd, dtype=torch.float32)
15 x_te = torch.tensor((X[test] - mean) / sd, dtype=torch.float32)
16 y_tr = torch.tensor(labels[train])          # class numbers 0, 1, 2 (integers)
17 y_te = torch.tensor(labels[test])
18
19 torch.manual_seed(0)
20 model = nn.Sequential(nn.Linear(4, 16), nn.ReLU(), nn.Linear(16, 3)) # 3 scores
    ↪ out
21 loss_fn = nn.CrossEntropyLoss()           # applies softmax, then -ln p(true
    ↪ class)
22 optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
23
24 for epoch in range(1, 31):
25     perm = torch.randperm(len(y_tr))      # shuffle, then mini-batches of 32
26     for start in range(0, len(y_tr), 32):
27         batch = perm[start:start + 32]
28         optimizer.zero_grad()
29         loss = loss_fn(model(x_tr[batch]), y_tr[batch])
30         loss.backward()
31         optimizer.step()
32     if epoch in (1, 5, 30):
33         with torch.no_grad():
34             acc = (model(x_te).argmax(dim=1) == y_te).float().mean().item()
35             print(f"epoch {epoch:2d}: training loss {loss_fn(model(x_tr),
    ↪ y_tr).item():.3f},"
36                   f" test accuracy {acc:.3f}")
37
38 with torch.no_grad():
39     probs = torch.softmax(model(x_te), dim=1)
40 predicted = probs.argmax(dim=1)
41 matrix = np.zeros((3, 3), dtype=int)
42 for p, a in zip(predicted, y_te):
43     matrix[p, a] += 1
44 print("Rows: predicted, columns: actual", names)
45 print(matrix)
46 print("Least confident test penguin:")
47 i = probs.max(dim=1).values.argmax()
48 print(" probabilities", np.round(probs[i].numpy(), 3), " actually", names[y_te[i]])

```

```

epoch 1: training loss 0.757, test accuracy 0.800
epoch 5: training loss 0.119, test accuracy 0.970
epoch 30: training loss 0.015, test accuracy 0.990
Rows: predicted, columns: actual ['Adelie' 'Chinstrap' 'Gentoo']
[[42  1  0]
 [ 0 25  0]
 [ 0  0 32]]
Least confident test penguin:
probabilities [0.325 0.675 0.001] actually Chinstrap

```

A few details:

- The network's last layer has no activation. `nn.CrossEntropyLoss` takes the raw logits and applies the softmax itself, which is more accurate numerically than taking the log of a softmax you computed separately. When you want the probabilities, call `torch.softmax` yourself.
- The labels are integer class numbers, not one-hot vectors; the loss function picks out the probability of the correct class.
- `argmax(dim=1)` finds the largest logit in each row, which is the predicted class.
- `torch.randperm` shuffles the training penguins each epoch, so the mini-batches are different every time.

The network gets 99 of the 100 test penguins right after 30 epochs. Its least confident prediction is a Chinstrap that it called Chinstrap with only 67.5% confidence, giving the rest to Adelie. The probabilities are useful in their own right: a researcher could double-check any penguin the network is unsure about.

Method (from this unit)	Test accuracy, species
k-nearest neighbors (2 features, 67 test penguins)	92.5%
decision tree (4 features, 67 test penguins)	95.5%
random forest (4 features, 67 test penguins)	97.0%
neural network (4 features, 100 test penguins)	99.0%

The test sets are different sizes and the features differ, so the table is a rough comparison, not a contest. On this data, every reasonable method does well.

2.4 Bending the Boundary

Without a hidden layer, a softmax network is *softmax regression*, the multi-class version of logistic regression, and its boundaries between classes are straight lines. Hidden layers let the boundaries bend. Figure 2.1 shows both, trained on bill length and depth for all 333 penguins.

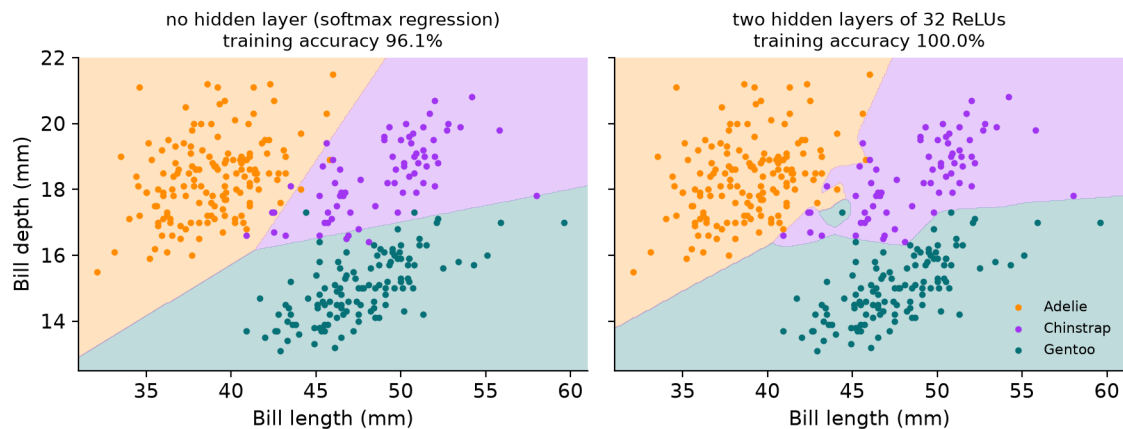


Figure 2.1: Predicted species across the plane. Left: no hidden layer. Right: two hidden layers of 32 ReLU neurons, trained for 1500 full-batch steps.

Straight boundaries already get 96.1% of the penguins right. The bigger network gets every training penguin right, but look at how: it grows small bays and islands to capture individual penguins in the overlap between Adelies, Chinstraps, and Gentoos. That is overfitting, the same thing a deep decision tree or 1-nearest neighbor did. The cures are the ones from the last chapter: a validation set with early stopping, or a smaller network.

Exercise 7 Compute a probability

A neuron has $z = 0$. What is the predicted probability of class 1?

Working Space

Answer on Page 38

Exercise 8 Loss of a wrong prediction

A network predicts $p = 0.1$ for an example whose true label is 1. What is the cross-entropy loss?

Working Space

Answer on Page 38

Exercise 9 Softmax by hand

Find the softmax probabilities for the logits $(1, 1, 3)$. What is the loss if the correct class is the first one?

Working Space

Answer on Page 38

Exercise 10 Shift the logits

Show that softmax gives the same probabilities for the logits $(2, 1, 0)$ as for $(12, 11, 10)$. Why do programs subtract the largest logit before exponentiating?

Working Space

Answer on Page 38

Exercise 11 Two features only

Change `species_net.py` to use only bill length and bill depth. How does the test accuracy after 30 epochs compare with k-nearest neighbors on the same two features?

Working Space

Answer on Page 38

Deep Learning

The networks in the last two chapters had one or two hidden layers. The systems that recognize your voice, translate languages, and generate images have dozens or hundreds. This chapter explains what all those layers buy you, the one idea from calculus that makes it possible to train them, and the special kinds of layers that let networks see and read.

3.1 Many Layers

A *deep* neural network has many hidden layers. Each layer builds features from the layer before it, so the features get more abstract as you go up. In a network that recognizes faces, the first layers respond to edges, middle layers combine edges into eyes and noses, and late layers combine those into whole faces. Nobody programs these features; the network learns them, because they help reduce the loss.

Two things had to happen before deep networks worked well: enough data and computing power to train them, and answers to the question of how to get a useful gradient to a weight that is fifty layers away from the loss. The second is the subject of the next three sections.

3.2 The Chain Rule

Training a network requires derivatives of a loss through many nested functions. The *chain rule* gives the derivative of a composition: if $y = f(g(x))$, then

$$\frac{dy}{dx} = f'(g(x)) \cdot g'(x)$$

For example, let $y = \sin(x^2 + 1)$. Here $f(u) = \sin u$ and $g(x) = x^2 + 1$, so

$$\frac{dy}{dx} = \cos(x^2 + 1) \cdot 2x$$

At $x = 1.3$, this is $2(1.3) \cos(2.69) \approx -2.339$. With more layers, the chain just gets longer: for $y = f(g(h(x)))$, the derivative is $f'(g(h(x))) \cdot g'(h(x)) \cdot h'(x)$.

3.3 Backpropagation

Backpropagation applies the chain rule to every weight in the network. A forward pass computes the output and the loss, remembering every intermediate value. A backward pass then starts at the loss and moves toward the input, multiplying local derivatives and reusing the values computed in the forward pass.

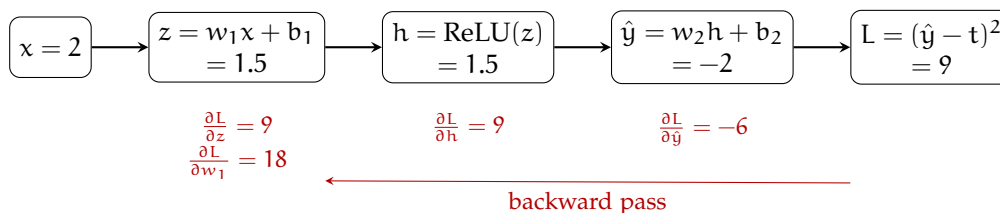


Figure 3.1: A forward pass (black) and a backward pass (red) through a tiny network with $w_1 = 0.5$, $b_1 = 0.5$, $w_2 = -1.5$, $b_2 = 0.25$, input $x = 2$, and target $t = 1$.

Example: For the network in Figure 3.1, find the gradient of the loss with respect to every parameter.

Solution: The forward pass gives $z = 0.5(2) + 0.5 = 1.5$, $h = 1.5$, $\hat{y} = -1.5(1.5) + 0.25 = -2$, and $L = (-2 - 1)^2 = 9$. Now go backward, one link at a time:

$$\begin{aligned}
 \frac{\partial L}{\partial \hat{y}} &= 2(\hat{y} - t) = -6 \\
 \frac{\partial L}{\partial w_2} &= \frac{\partial L}{\partial \hat{y}} \cdot h = -9, & \frac{\partial L}{\partial b_2} &= \frac{\partial L}{\partial \hat{y}} \cdot 1 = -6 \\
 \frac{\partial L}{\partial h} &= \frac{\partial L}{\partial \hat{y}} \cdot w_2 = 9 \\
 \frac{\partial L}{\partial z} &= \frac{\partial L}{\partial h} \cdot \text{ReLU}'(z) = 9 \cdot 1 = 9 & (\text{since } z > 0) \\
 \frac{\partial L}{\partial w_1} &= \frac{\partial L}{\partial z} \cdot x = 18, & \frac{\partial L}{\partial b_1} &= \frac{\partial L}{\partial z} \cdot 1 = 9
 \end{aligned}$$

Notice how each line reuses the one above it. In a real network, $\frac{\partial L}{\partial h}$ is computed once and shared by every weight below h . That sharing is what makes backpropagation fast: one backward sweep costs about as much as one forward pass, no matter how many weights there are.

The program `backprop.py` does the calculation by hand and then lets PyTorch do it:

```

1 import torch
2
3 # A tiny network: h = relu(w1*x + b1), y_hat = w2*h + b2, loss = (y_hat - t)^2
4 x, t = 2.0, 1.0
5 w1, b1, w2, b2 = 0.5, 0.5, -1.5, 0.25

```

```

6
7 # Forward pass by hand
8 z = w1 * x + b1          # 1.5
9 h = max(0.0, z)          # 1.5
10 y_hat = w2 * h + b2      # -2.0
11 loss = (y_hat - t)**2     # 9.0
12
13 # Backward pass by hand: the chain rule, from the loss back toward the input
14 dL_dyhat = 2 * (y_hat - t) # -6
15 dL_dw2 = dL_dyhat * h      # -9
16 dL_db2 = dL_dyhat * 1      # -6
17 dL_dh = dL_dyhat * w2      # 9
18 dL_dz = dL_dh * (1.0 if z > 0 else 0.0) # ReLU passes the gradient only if z > 0
19 dL_dw1 = dL_dz * x         # 18
20 dL_db1 = dL_dz * 1         # 9
21 print("by hand: ", loss, [dL_dw1, dL_db1, dL_dw2, dL_db2])
22
23 # The same thing with PyTorch's automatic differentiation
24 params = [torch.tensor(v, requires_grad=True) for v in (w1, b1, w2, b2)]
25 W1, B1, W2, B2 = params
26 loss_t = (W2 * torch.relu(W1 * x + B1) + B2 - t)**2
27 loss_t.backward()
28 print("autograd: ", loss_t.item(), [p.grad.item() for p in params])

```

```

by hand:  9.0 [18.0, 9.0, -9.0, -6.0]
autograd: 9.0 [18.0, 9.0, -9.0, -6.0]

```

They agree. When you call `loss.backward()` in PyTorch, this is exactly what happens, for every weight.

3.4 Vanishing Gradients

The gradient for a weight deep inside a network is a long product of local derivatives, one or more per layer. If those factors are usually less than 1, the product shrinks exponentially; if they are usually more than 1, it explodes. The sigmoid's derivative is never more than $\frac{1}{4}$, so a gradient passing back through many sigmoid layers fades away, and the early layers stop learning.

The program `vanishing.py` builds networks of different depths, each layer with 64 neurons, and measures the average size of the gradient that reaches the first layer:

```

1 import torch
2 from torch import nn
3
4 def first_layer_gradient(activation, depth, seed=0):
5     """Average size of the gradient reaching the first layer of a deep network."""

```

```

6 torch.manual_seed(seed)
7 layers = []
8 for _ in range(depth):
9     linear = nn.Linear(64, 64)
10    if activation is nn.ReLU:      # "He" initialization, designed for ReLU
11        nn.init.kaiming_normal_(linear.weight, nonlinearity="relu")
12    else:                          # "Xavier" initialization, designed for
13        sigmoid-like units
14        nn.init.xavier_normal_(linear.weight)
15        nn.init.zeros_(linear.bias)
16        layers += [linear, activation()]
17    model = nn.Sequential(*layers, nn.Linear(64, 1))
18    x = torch.randn(256, 64)
19    loss = model(x).pow(2).mean()
20    loss.backward()
21    return model[0].weight.grad.abs().mean().item()
22
23 print("depth    sigmoid    ReLU")
24 for depth in [2, 5, 10, 20, 40]:
25     s = first_layer_gradient(nn.Sigmoid, depth)
26     r = first_layer_gradient(nn.ReLU, depth)
27     print(f"depth:5d} {s:9.2e} {r:9.2e}")

```

depth	sigmoid	ReLU
2	4.11e-05	2.97e-03
5	1.11e-06	7.26e-03
10	7.34e-10	1.17e-03
20	6.90e-16	6.92e-04
40	4.56e-29	1.58e-04

With sigmoids, the gradient at the first layer of a 40-layer network is around 10^{-29} : those weights will never move. With ReLUs, whose slope is exactly 1 wherever they are active, and with starting weights scaled carefully for them (the “He” initialization), the gradient stays within a factor of about 50 at every depth. Better activations and careful initialization, along with tricks such as *residual connections*, which let the signal skip around layers, are what made networks with hundreds of layers trainable.

3.5 Convolutions: Networks That See

A 256×256 color photo has $256 \times 256 \times 3 = 196,608$ numbers. A dense layer of 1000 neurons connected to all of them would need about 197 million weights, for one layer, and it would have to learn separately that an eye in the top left corner looks like an eye in the bottom right.

A *convolutional layer* instead slides a small *filter*, often 3×3 , across the image. At each position, it multiplies the filter by the patch of pixels under it and adds up the products. The same nine weights are used everywhere in the image, so the layer finds the same

pattern wherever it appears.

Example: Apply the filter

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \quad \text{to the patch} \quad \begin{bmatrix} 0 & 0 & 9 \\ 0 & 0 & 9 \\ 0 & 0 & 9 \end{bmatrix}$$

Solution: Multiply matching entries and add: the left column contributes $-(0+0+0) = 0$, the middle column contributes 0, and the right column contributes $9 + 9 + 9 = 27$. The output is 27. On a patch that is all the same brightness, the left and right columns cancel, and the output is 0. So this filter lights up where the image gets brighter from left to right: it detects vertical edges.

`convolution.py` slides the filter across a small image with SciPy:

```

1 import numpy as np
2 from scipy.signal import correlate2d
3
4 # A 5 x 6 grayscale "image": dark on the left, bright on the right
5 image = np.array([[0, 0, 0, 9, 9, 9],
6                  [0, 0, 0, 9, 9, 9],
7                  [0, 0, 0, 9, 9, 9],
8                  [0, 0, 0, 9, 9, 9],
9                  [0, 0, 0, 9, 9, 9]])
10
11 # A 3 x 3 filter that responds to a change from left to right
12 vertical_edge = np.array([[-1, 0, 1],
13                          [-1, 0, 1],
14                          [-1, 0, 1]])
15
16 # Slide the filter over every 3 x 3 patch; each output is a sum of products
17 print(correlate2d(image, vertical_edge, mode="valid"))
18
19 horizontal_edge = vertical_edge.T
20 print(correlate2d(image, horizontal_edge, mode="valid"))

```

```

[[ 0 27 27 0]
 [ 0 27 27 0]
 [ 0 27 27 0]]
[[0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]]

```

A 3×3 filter fits in 3×4 positions on a 5×6 image, so the output is 3×4 . The vertical-edge filter finds the boundary in the middle columns; the horizontal-edge filter (its transpose) finds nothing, because nothing changes from top to bottom. (The function is called

`correlate2d` because, strictly speaking, a mathematician's convolution flips the filter first. Deep learning libraries do not flip, but they still call it convolution.)

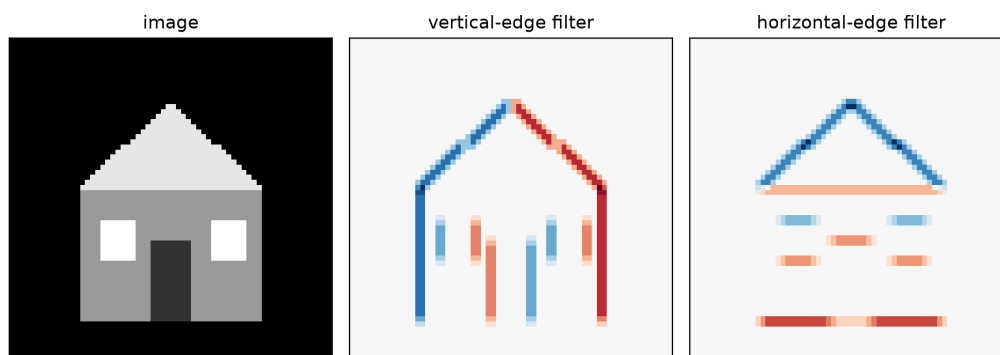


Figure 3.2: The two edge filters applied to a 64×64 drawing. Red is a positive response, blue a negative one.

In a *convolutional neural network* (CNN), nobody chooses the filters. Each layer has dozens of them, their nine weights are learned by backpropagation like any other weights, and the first layers usually end up looking a lot like these edge detectors. Later layers apply filters to the outputs of earlier ones, finding corners, textures, and eventually whole objects.

3.6 A Field Guide to Architectures

Architecture	Key idea	Typical uses
dense (fully connected)	every neuron sees every input	tables of numbers, like the penguins
convolutional (CNN)	small filters shared across the image	photos, medical scans
recurrent (RNN)	reads a sequence one step at a time, with memory	older speech and text systems
transformer	<i>attention</i> : each word looks at every other word to decide what matters	chatbots, translation

The large language models behind chatbots are transformers with billions of weights, trained by the same loop you wrote for the penguins: feedforward, cross-entropy loss on the next word, backpropagation, and a gradient step, repeated across a vast amount of text.

Exercise 12 **Apply the chain rule**

Find $\frac{d}{dx}(3x - 1)^4$.

Working Space

Answer on Page 39

Exercise 13 **One gradient**

A single weight has $\hat{y} = wx$ and $L = (\hat{y} - t)^2$. Find $\frac{\partial L}{\partial w}$ when $w = 2$, $x = 3$, and $t = 5$.

Working Space

Answer on Page 39

Exercise 14 **Vanishing gradients**

The sigmoid activation has a derivative of at most 0.25. A gradient passes back through 10 sigmoid layers, picking up a factor of at most 0.25 at each layer. What is the largest the resulting factor can be?

Working Space

Answer on Page 39

Exercise 15 **A dead ReLU**

In the network of Figure 3.1, change b_1 to -1.5 , so $z = -0.5$. Redo the backward pass. Which gradients become zero, and

Working Space

why?

Answer on Page 39

Exercise 16 **Filter by hand**

Apply the horizontal-edge filter (the transpose of the vertical-edge filter) to

the patch $\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 5 & 5 & 5 \end{bmatrix}$.

Working Space

Answer on Page 39

Exercise 17 **Count the weights**

A convolutional layer applies 32 different 3×3 filters to a grayscale image, each with one bias. How many parameters does it have? How many would a dense layer of 32 neurons connected to every

Working Space

pixel of a 256×256 grayscale image have?

Answer on Page 39

Statistical Tables

This chapter collects the three tables printed on the AP Statistics formula sheet: standard normal probabilities, t critical values, and chi-square critical values. The layouts match the exam versions, so practice with these tables carries over directly to test day.

4.1 The Standard Normal Table

Table 4.1 and Table 4.2 give the *cumulative probability* $P(Z \leq z)$, the area under the standard normal curve to the left of z (Figure 4.1). Each row gives z to one decimal place, and each column adds the second decimal place. To find $P(Z \leq 1.37)$, go to row 1.3 and column .07; the entry is 0.9147.

Three rules cover every question this table answers:

- Area to the left of z : read the table directly.
- Area to the right of z : subtract the table value from 1.
- Area between z_1 and z_2 : subtract the smaller table value from the larger one.

To work backward from an area to a z -score, find the entry closest to the area in the body of the table, then read its row and column.

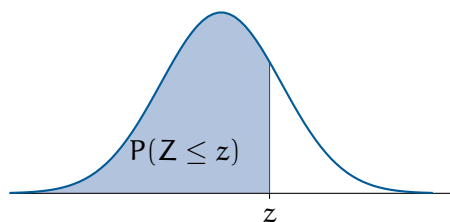


Figure 4.1: Each entry in the standard normal table is the shaded area to the left of z .

Table 4.1: Standard normal cumulative probabilities, $P(Z \leq z)$, negative z

z	.00	.01	.02	.03	.04	.05	.06	.07	.08	.09
-3.4	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003	0.0002
-3.3	0.0005	0.0005	0.0005	0.0004	0.0004	0.0004	0.0004	0.0004	0.0004	0.0003
-3.2	0.0007	0.0007	0.0006	0.0006	0.0006	0.0006	0.0006	0.0005	0.0005	0.0005
-3.1	0.0010	0.0009	0.0009	0.0009	0.0008	0.0008	0.0008	0.0008	0.0007	0.0007
-3.0	0.0013	0.0013	0.0013	0.0012	0.0012	0.0011	0.0011	0.0011	0.0010	0.0010
-2.9	0.0019	0.0018	0.0018	0.0017	0.0016	0.0016	0.0015	0.0015	0.0014	0.0014
-2.8	0.0026	0.0025	0.0024	0.0023	0.0023	0.0022	0.0021	0.0021	0.0020	0.0019
-2.7	0.0035	0.0034	0.0033	0.0032	0.0031	0.0030	0.0029	0.0028	0.0027	0.0026
-2.6	0.0047	0.0045	0.0044	0.0043	0.0041	0.0040	0.0039	0.0038	0.0037	0.0036
-2.5	0.0062	0.0060	0.0059	0.0057	0.0055	0.0054	0.0052	0.0051	0.0049	0.0048
-2.4	0.0082	0.0080	0.0078	0.0075	0.0073	0.0071	0.0069	0.0068	0.0066	0.0064
-2.3	0.0107	0.0104	0.0102	0.0099	0.0096	0.0094	0.0091	0.0089	0.0087	0.0084
-2.2	0.0139	0.0136	0.0132	0.0129	0.0125	0.0122	0.0119	0.0116	0.0113	0.0110
-2.1	0.0179	0.0174	0.0170	0.0166	0.0162	0.0158	0.0154	0.0150	0.0146	0.0143
-2.0	0.0228	0.0222	0.0217	0.0212	0.0207	0.0202	0.0197	0.0192	0.0188	0.0183
-1.9	0.0287	0.0281	0.0274	0.0268	0.0262	0.0256	0.0250	0.0244	0.0239	0.0233
-1.8	0.0359	0.0351	0.0344	0.0336	0.0329	0.0322	0.0314	0.0307	0.0301	0.0294
-1.7	0.0446	0.0436	0.0427	0.0418	0.0409	0.0401	0.0392	0.0384	0.0375	0.0367
-1.6	0.0548	0.0537	0.0526	0.0516	0.0505	0.0495	0.0485	0.0475	0.0465	0.0455
-1.5	0.0668	0.0655	0.0643	0.0630	0.0618	0.0606	0.0594	0.0582	0.0571	0.0559
-1.4	0.0808	0.0793	0.0778	0.0764	0.0749	0.0735	0.0721	0.0708	0.0694	0.0681
-1.3	0.0968	0.0951	0.0934	0.0918	0.0901	0.0885	0.0869	0.0853	0.0838	0.0823
-1.2	0.1151	0.1131	0.1112	0.1093	0.1075	0.1056	0.1038	0.1020	0.1003	0.0985
-1.1	0.1357	0.1335	0.1314	0.1292	0.1271	0.1251	0.1230	0.1210	0.1190	0.1170
-1.0	0.1587	0.1562	0.1539	0.1515	0.1492	0.1469	0.1446	0.1423	0.1401	0.1379
-0.9	0.1841	0.1814	0.1788	0.1762	0.1736	0.1711	0.1685	0.1660	0.1635	0.1611
-0.8	0.2119	0.2090	0.2061	0.2033	0.2005	0.1977	0.1949	0.1922	0.1894	0.1867
-0.7	0.2420	0.2389	0.2358	0.2327	0.2296	0.2266	0.2236	0.2206	0.2177	0.2148
-0.6	0.2743	0.2709	0.2676	0.2643	0.2611	0.2578	0.2546	0.2514	0.2483	0.2451
-0.5	0.3085	0.3050	0.3015	0.2981	0.2946	0.2912	0.2877	0.2843	0.2810	0.2776
-0.4	0.3446	0.3409	0.3372	0.3336	0.3300	0.3264	0.3228	0.3192	0.3156	0.3121
-0.3	0.3821	0.3783	0.3745	0.3707	0.3669	0.3632	0.3594	0.3557	0.3520	0.3483
-0.2	0.4207	0.4168	0.4129	0.4090	0.4052	0.4013	0.3974	0.3936	0.3897	0.3859
-0.1	0.4602	0.4562	0.4522	0.4483	0.4443	0.4404	0.4364	0.4325	0.4286	0.4247
-0.0	0.5000	0.4960	0.4920	0.4880	0.4840	0.4801	0.4761	0.4721	0.4681	0.4641

Table 4.2: Standard normal cumulative probabilities, $P(Z \leq z)$, positive z

z	.00	.01	.02	.03	.04	.05	.06	.07	.08	.09
0.0	0.5000	0.5040	0.5080	0.5120	0.5160	0.5199	0.5239	0.5279	0.5319	0.5359
0.1	0.5398	0.5438	0.5478	0.5517	0.5557	0.5596	0.5636	0.5675	0.5714	0.5753
0.2	0.5793	0.5832	0.5871	0.5910	0.5948	0.5987	0.6026	0.6064	0.6103	0.6141
0.3	0.6179	0.6217	0.6255	0.6293	0.6331	0.6368	0.6406	0.6443	0.6480	0.6517
0.4	0.6554	0.6591	0.6628	0.6664	0.6700	0.6736	0.6772	0.6808	0.6844	0.6879
0.5	0.6915	0.6950	0.6985	0.7019	0.7054	0.7088	0.7123	0.7157	0.7190	0.7224
0.6	0.7257	0.7291	0.7324	0.7357	0.7389	0.7422	0.7454	0.7486	0.7517	0.7549
0.7	0.7580	0.7611	0.7642	0.7673	0.7704	0.7734	0.7764	0.7794	0.7823	0.7852
0.8	0.7881	0.7910	0.7939	0.7967	0.7995	0.8023	0.8051	0.8078	0.8106	0.8133
0.9	0.8159	0.8186	0.8212	0.8238	0.8264	0.8289	0.8315	0.8340	0.8365	0.8389
1.0	0.8413	0.8438	0.8461	0.8485	0.8508	0.8531	0.8554	0.8577	0.8599	0.8621
1.1	0.8643	0.8665	0.8686	0.8708	0.8729	0.8749	0.8770	0.8790	0.8810	0.8830
1.2	0.8849	0.8869	0.8888	0.8907	0.8925	0.8944	0.8962	0.8980	0.8997	0.9015
1.3	0.9032	0.9049	0.9066	0.9082	0.9099	0.9115	0.9131	0.9147	0.9162	0.9177
1.4	0.9192	0.9207	0.9222	0.9236	0.9251	0.9265	0.9279	0.9292	0.9306	0.9319
1.5	0.9332	0.9345	0.9357	0.9370	0.9382	0.9394	0.9406	0.9418	0.9429	0.9441
1.6	0.9452	0.9463	0.9474	0.9484	0.9495	0.9505	0.9515	0.9525	0.9535	0.9545
1.7	0.9554	0.9564	0.9573	0.9582	0.9591	0.9599	0.9608	0.9616	0.9625	0.9633
1.8	0.9641	0.9649	0.9656	0.9664	0.9671	0.9678	0.9686	0.9693	0.9699	0.9706
1.9	0.9713	0.9719	0.9726	0.9732	0.9738	0.9744	0.9750	0.9756	0.9761	0.9767
2.0	0.9772	0.9778	0.9783	0.9788	0.9793	0.9798	0.9803	0.9808	0.9812	0.9817
2.1	0.9821	0.9826	0.9830	0.9834	0.9838	0.9842	0.9846	0.9850	0.9854	0.9857
2.2	0.9861	0.9864	0.9868	0.9871	0.9875	0.9878	0.9881	0.9884	0.9887	0.9890
2.3	0.9893	0.9896	0.9898	0.9901	0.9904	0.9906	0.9909	0.9911	0.9913	0.9916
2.4	0.9918	0.9920	0.9922	0.9925	0.9927	0.9929	0.9931	0.9932	0.9934	0.9936
2.5	0.9938	0.9940	0.9941	0.9943	0.9945	0.9946	0.9948	0.9949	0.9951	0.9952
2.6	0.9953	0.9955	0.9956	0.9957	0.9959	0.9960	0.9961	0.9962	0.9963	0.9964
2.7	0.9965	0.9966	0.9967	0.9968	0.9969	0.9970	0.9971	0.9972	0.9973	0.9974
2.8	0.9974	0.9975	0.9976	0.9977	0.9977	0.9978	0.9979	0.9979	0.9980	0.9981
2.9	0.9981	0.9982	0.9982	0.9983	0.9984	0.9984	0.9985	0.9985	0.9986	0.9986
3.0	0.9987	0.9987	0.9987	0.9988	0.9988	0.9989	0.9989	0.9989	0.9990	0.9990
3.1	0.9990	0.9991	0.9991	0.9991	0.9992	0.9992	0.9992	0.9992	0.9993	0.9993
3.2	0.9993	0.9993	0.9994	0.9994	0.9994	0.9994	0.9994	0.9995	0.9995	0.9995
3.3	0.9995	0.9995	0.9995	0.9996	0.9996	0.9996	0.9996	0.9996	0.9996	0.9997
3.4	0.9997	0.9997	0.9997	0.9997	0.9997	0.9997	0.9997	0.9997	0.9997	0.9998

Exercise 18 Areas under the standard normal curve

Use the standard normal table to find each probability.

Working Space

- (a) $P(Z \leq 1.37)$
 (b) $P(Z > -0.84)$

(c) $P(-1.25 < Z < 0.60)$

Answer on Page 39

Exercise 19 From an area back to z

Find the z -score that has an area of 0.90 to its left.

Working Space

Answer on Page 40

4.2 The t Table

Table 4.3 gives the critical value t^* that has upper-tail probability p for a t distribution with df degrees of freedom (Figure 4.2). The table is read two ways.

- For a confidence interval, use the confidence level C along the bottom row. A 95 percent interval with $df = 24$ uses the column above 95%, giving $t^* = 2.064$.
- For a significance test, locate the test statistic between two entries in its df row. The tail probabilities at the top of those two columns bound the one-sided P -value. Double both bounds for a two-sided test.

When the exact df is missing from the table, use the next smaller df that appears. The smaller df gives a slightly larger t^* , so the resulting interval is a little wider and the conclusion stays conservative. The last row, $df = \infty$, contains the standard normal critical values z^* .

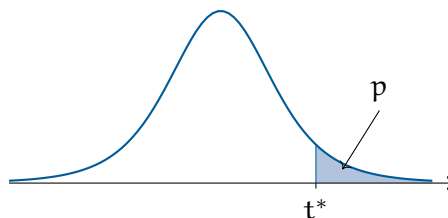


Figure 4.2: Each entry in the t table is the value t^* that leaves upper-tail probability p .

Table 4.3: t distribution critical values, t^* with upper-tail probability p

df	Tail probability p											
	.25	.20	.15	.10	.05	.025	.02	.01	.005	.0025	.001	.0005
1	1.000	1.376	1.963	3.078	6.314	12.706	15.895	31.821	63.657	127.321	318.309	636.619
2	0.816	1.061	1.386	1.886	2.920	4.303	4.849	6.965	9.925	14.089	22.327	31.599
3	0.765	0.978	1.250	1.638	2.353	3.182	3.482	4.541	5.841	7.453	10.215	12.924
4	0.741	0.941	1.190	1.533	2.132	2.776	2.999	3.747	4.604	5.598	7.173	8.610
5	0.727	0.920	1.156	1.476	2.015	2.571	2.757	3.365	4.032	4.773	5.893	6.869
6	0.718	0.906	1.134	1.440	1.943	2.447	2.612	3.143	3.707	4.317	5.208	5.959
7	0.711	0.896	1.119	1.415	1.895	2.365	2.517	2.998	3.499	4.029	4.785	5.408
8	0.706	0.889	1.108	1.397	1.860	2.306	2.449	2.896	3.355	3.833	4.501	5.041
9	0.703	0.883	1.100	1.383	1.833	2.262	2.398	2.821	3.250	3.690	4.297	4.781
10	0.700	0.879	1.093	1.372	1.812	2.228	2.359	2.764	3.169	3.581	4.144	4.587
11	0.697	0.876	1.088	1.363	1.796	2.201	2.328	2.718	3.106	3.497	4.025	4.437
12	0.695	0.873	1.083	1.356	1.782	2.179	2.303	2.681	3.055	3.428	3.930	4.318
13	0.694	0.870	1.079	1.350	1.771	2.160	2.282	2.650	3.012	3.372	3.852	4.221
14	0.692	0.868	1.076	1.345	1.761	2.145	2.264	2.624	2.977	3.326	3.787	4.140
15	0.691	0.866	1.074	1.341	1.753	2.131	2.249	2.602	2.947	3.286	3.733	4.073
16	0.690	0.865	1.071	1.337	1.746	2.120	2.235	2.583	2.921	3.252	3.686	4.015
17	0.689	0.863	1.069	1.333	1.740	2.110	2.224	2.567	2.898	3.222	3.646	3.965
18	0.688	0.862	1.067	1.330	1.734	2.101	2.214	2.552	2.878	3.197	3.610	3.922
19	0.688	0.861	1.066	1.328	1.729	2.093	2.205	2.539	2.861	3.174	3.579	3.883
20	0.687	0.860	1.064	1.325	1.725	2.086	2.197	2.528	2.845	3.153	3.552	3.850
21	0.686	0.859	1.063	1.323	1.721	2.080	2.189	2.518	2.831	3.135	3.527	3.819
22	0.686	0.858	1.061	1.321	1.717	2.074	2.183	2.508	2.819	3.119	3.505	3.792
23	0.685	0.858	1.060	1.319	1.714	2.069	2.177	2.500	2.807	3.104	3.485	3.768
24	0.685	0.857	1.059	1.318	1.711	2.064	2.172	2.492	2.797	3.091	3.467	3.745
25	0.684	0.856	1.058	1.316	1.708	2.060	2.167	2.485	2.787	3.078	3.450	3.725
26	0.684	0.856	1.058	1.315	1.706	2.056	2.162	2.479	2.779	3.067	3.435	3.707
27	0.684	0.855	1.057	1.314	1.703	2.052	2.158	2.473	2.771	3.057	3.421	3.690
28	0.683	0.855	1.056	1.313	1.701	2.048	2.154	2.467	2.763	3.047	3.408	3.674
29	0.683	0.854	1.055	1.311	1.699	2.045	2.150	2.462	2.756	3.038	3.396	3.659
30	0.683	0.854	1.055	1.310	1.697	2.042	2.147	2.457	2.750	3.030	3.385	3.646
40	0.681	0.851	1.050	1.303	1.684	2.021	2.123	2.423	2.704	2.971	3.307	3.551
50	0.679	0.849	1.047	1.299	1.676	2.009	2.109	2.403	2.678	2.937	3.261	3.496
60	0.679	0.848	1.045	1.296	1.671	2.000	2.099	2.390	2.660	2.915	3.232	3.460
80	0.678	0.846	1.043	1.292	1.664	1.990	2.088	2.374	2.639	2.887	3.195	3.416
100	0.677	0.845	1.042	1.290	1.660	1.984	2.081	2.364	2.626	2.871	3.174	3.390
1000	0.675	0.842	1.037	1.282	1.646	1.962	2.056	2.330	2.581	2.813	3.098	3.300
∞	0.674	0.842	1.036	1.282	1.645	1.960	2.054	2.326	2.576	2.807	3.090	3.291
	50%	60%	70%	80%	90%	95%	96%	98%	99%	99.5%	99.8%	99.9%
	Confidence level C											

Exercise 20 Critical values for confidence intervalsFind t^* for each interval.

Working Space

- (a) A 95 percent confidence interval with $df = 24$.
- (b) A 99 percent confidence interval with

df = 45.

Answer on Page 40

Exercise 21 Bounding a P-value

A one-sided test with $df = 15$ produces $t = 2.30$. Use the t table to find bounds on the P-value.

Working Space

Answer on Page 40

4.3 The Chi-Square Table

Table 4.4 gives the critical value χ^2 that has upper-tail probability p for a chi-square distribution with df degrees of freedom (Figure 4.3). It extends the shorter table in the chi-square chapter to more tail areas and to $df = 100$. Chi-square tests use only the upper tail, so the P-value bounds come straight from the column headings with no doubling.

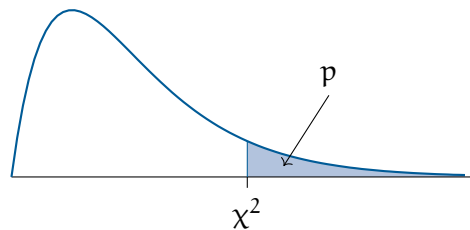


Figure 4.3: Each entry in the chi-square table is the value χ^2 that leaves upper-tail probability p .

Table 4.4: Chi-square critical values, χ^2 with upper-tail probability p

df	Tail probability p											
	.25	.20	.15	.10	.05	.025	.02	.01	.005	.0025	.001	.0005
1	1.323	1.642	2.072	2.706	3.841	5.024	5.412	6.635	7.879	9.141	10.828	12.116
2	2.773	3.219	3.794	4.605	5.991	7.378	7.824	9.210	10.597	11.983	13.816	15.202
3	4.108	4.642	5.317	6.251	7.815	9.348	9.837	11.345	12.838	14.320	16.266	17.730
4	5.385	5.989	6.745	7.779	9.488	11.143	11.668	13.277	14.860	16.424	18.467	19.997
5	6.626	7.289	8.115	9.236	11.070	12.833	13.388	15.086	16.750	18.386	20.515	22.105
6	7.841	8.558	9.446	10.645	12.592	14.449	15.033	16.812	18.548	20.249	22.458	24.103
7	9.037	9.803	10.748	12.017	14.067	16.013	16.622	18.475	20.278	22.040	24.322	26.018
8	10.219	11.030	12.027	13.362	15.507	17.535	18.168	20.090	21.955	23.774	26.124	27.868
9	11.389	12.242	13.288	14.684	16.919	19.023	19.679	21.666	23.589	25.462	27.877	29.666
10	12.549	13.442	14.534	15.987	18.307	20.483	21.161	23.209	25.188	27.112	29.588	31.420
11	13.701	14.631	15.767	17.275	19.675	21.920	22.618	24.725	26.757	28.729	31.264	33.137
12	14.845	15.812	16.989	18.549	21.026	23.337	24.054	26.217	28.300	30.318	32.909	34.821
13	15.984	16.985	18.202	19.812	22.362	24.736	25.472	27.688	29.819	31.883	34.528	36.478
14	17.117	18.151	19.406	21.064	23.685	26.119	26.873	29.141	31.319	33.426	36.123	38.109
15	18.245	19.311	20.603	22.307	24.996	27.488	28.259	30.578	32.801	34.950	37.697	39.719
16	19.369	20.465	21.793	23.542	26.296	28.845	29.633	32.000	34.267	36.456	39.252	41.308
17	20.489	21.615	22.977	24.769	27.587	30.191	30.995	33.409	35.718	37.946	40.790	42.879
18	21.605	22.760	24.155	25.989	28.869	31.526	32.346	34.805	37.156	39.422	42.312	44.434
19	22.718	23.900	25.329	27.204	30.144	32.852	33.687	36.191	38.582	40.885	43.820	45.973
20	23.828	25.038	26.498	28.412	31.410	34.170	35.020	37.566	39.997	42.336	45.315	47.498
21	24.935	26.171	27.662	29.615	32.671	35.479	36.343	38.932	41.401	43.775	46.797	49.011
22	26.039	27.301	28.822	30.813	33.924	36.781	37.659	40.289	42.796	45.204	48.268	50.511
23	27.141	28.429	29.979	32.007	35.172	38.076	38.968	41.638	44.181	46.623	49.728	52.000
24	28.241	29.553	31.132	33.196	36.415	39.364	40.270	42.980	45.559	48.034	51.179	53.479
25	29.339	30.675	32.282	34.382	37.652	40.646	41.566	44.314	46.928	49.435	52.620	54.947
26	30.435	31.795	33.429	35.563	38.885	41.923	42.856	45.642	48.290	50.829	54.052	56.407
27	31.528	32.912	34.574	36.741	40.113	43.195	44.140	46.963	49.645	52.215	55.476	57.858
28	32.620	34.027	35.715	37.916	41.337	44.461	45.419	48.278	50.993	53.594	56.892	59.300
29	33.711	35.139	36.854	39.087	42.557	45.722	46.693	49.588	52.336	54.967	58.301	60.735
30	34.800	36.250	37.990	40.256	43.773	46.979	47.962	50.892	53.672	56.332	59.703	62.162
40	45.616	47.269	49.244	51.805	55.758	59.342	60.436	63.691	66.766	69.699	73.402	76.095
50	56.334	58.164	60.346	63.167	67.505	71.420	72.613	76.154	79.490	82.664	86.661	89.561
60	66.981	68.972	71.341	74.397	79.082	83.298	84.580	88.379	91.952	95.344	99.607	102.695
80	88.130	90.405	93.106	96.578	101.879	106.629	108.069	112.329	116.321	120.102	124.839	128.261
100	109.141	111.667	114.659	118.498	124.342	129.561	131.142	135.807	140.169	144.293	149.449	153.167

Exercise 22 Reading the chi-square table

Working Space

- Find the critical value for a chi-square test at $\alpha = 0.05$ with $df = 12$.
- A goodness-of-fit test with $df = 4$ produces $\chi^2 = 9.80$. Find bounds on

the P-value.

Answer on Page 40

Answers to Exercises

Answer to Exercise 1 (on page 10)

$\hat{y} = 5$, so the squared error is $(5 - 4)^2 = 1$.

Answer to Exercise 2 (on page 10)

The sigmoid output lies between 0 and 1, which would prevent the network from predicting values outside that range. Regression targets can be any real number, so the identity activation is used.

Answer to Exercise 3 (on page 10)

At $x = 0, 1, 2, 3, 4$: $h_1 = 0, 1, 2, 3, 4$ and $h_2 = 0, 0, 0, 1, 2$, so $\hat{y} = 0, 1, 2, 1, 0$. The output rises with slope 1 up to $x = 2$ and then falls with slope -1 : a tent (or upside-down V) peaking at $x = 2$. (For $x < 0$, the output is 0.)

Answer to Exercise 4 (on page 11)

First layer: $6 \times 32 + 32 = 224$. Second: $32 \times 32 + 32 = 1056$. Output: $32 + 1 = 33$. Total: $224 + 1056 + 33 = 1313$, as the program prints.

Answer to Exercise 5 (on page 11)

Substitute: $\hat{y} = W_2(W_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2 = (W_2W_1)\mathbf{x} + (W_2\mathbf{b}_1 + \mathbf{b}_2)$. That is one linear layer with weight matrix W_2W_1 and bias $W_2\mathbf{b}_1 + \mathbf{b}_2$. Without nonlinear activations, extra layers add parameters but no flexibility.

Answer to Exercise 6 (on page 11)

Most seeds end between about 0.68 and 0.96 degrees, but seed 2 gets stuck near 1.40 degrees. With that start, some hidden neurons are dead (their inputs are negative for every reading, so they never receive a gradient) and the rest cannot form enough bends. Exact numbers vary with the PyTorch version.

Answer to Exercise 7 (on page 17)

$$\sigma(0) = \frac{1}{1+1} = 0.5.$$

Answer to Exercise 8 (on page 17)

$-\ln(0.1) \approx 2.303$. The loss is large because the network gave the correct class a low probability.

Answer to Exercise 9 (on page 18)

$e^1 \approx 2.718$ (twice) and $e^3 \approx 20.086$, for a total of 25.522. The probabilities are about 0.107, 0.107, and 0.787. If the first class is correct, the loss is $-\ln 0.107 \approx 2.24$.

Answer to Exercise 10 (on page 18)

Adding 10 to every logit multiplies every e^{z_k} , and so the total, by e^{10} , which cancels in each fraction. Subtracting the largest logit therefore changes nothing mathematically, but it makes the largest exponent $e^0 = 1$, so huge logits like 1000 do not overflow to infinity. (We did the same trick for log probabilities in the Bayesian Classifiers chapter.)

Answer to Exercise 11 (on page 18)

Change the `np.column_stack` to the two bill columns and the first layer to `nn.Linear(2, 16)`. The test accuracy comes out at about 94% (exact numbers depend on the PyTorch version), similar to the 92.5% that k-nearest neighbors got on a different test split. With only two features, the overlap between Adelies and Chinstraps limits every method.

Answer to Exercise 12 (on page 25)

$$4(3x - 1)^3 \cdot 3 = 12(3x - 1)^3.$$

Answer to Exercise 13 (on page 25)

$$\hat{y} = 6, \text{ so } \frac{\partial L}{\partial w} = 2(6 - 5)(3) = 6.$$

Answer to Exercise 14 (on page 25)

$0.25^{10} \approx 9.5 \times 10^{-7}$. The gradient reaching the early layers is less than a millionth of its original size, so those layers barely learn. This is the vanishing gradient problem, and it is one reason modern networks use the ReLU, whose derivative is 1 for positive inputs.

Answer to Exercise 15 (on page 25)

Now $h = \text{ReLU}(-0.5) = 0$ and $\hat{y} = 0.25$, so $L = (0.25 - 1)^2 = 0.5625$ and $\frac{\partial L}{\partial \hat{y}} = -1.5$. Then $\frac{\partial L}{\partial w_2} = -1.5 \times 0 = 0$, $\frac{\partial L}{\partial b_2} = -1.5$, and since $\text{ReLU}'(z) = 0$ for $z < 0$, $\frac{\partial L}{\partial z} = 0$, so $\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial b_1} = 0$. Only b_2 can learn. If z is negative for every training example, the hidden neuron never changes: it is dead.

Answer to Exercise 16 (on page 26)

The horizontal-edge filter has rows $(-1, -1, -1)$, $(0, 0, 0)$, and $(1, 1, 1)$. The output is $-(1 + 1 + 1) + 0 + (5 + 5 + 5) = 12$: the patch gets brighter from top to bottom, so the filter responds.

Answer to Exercise 17 (on page 26)

Convolutional: $32 \times (9 + 1) = 320$, regardless of the image size. Dense: $32 \times (256 \times 256 + 1) = 2,097,184$. Sharing the filter's weights across the image saves a factor of more than 6000.

Answer to Exercise 18 (on page 31)

- (a) Row 1.3, column .07: 0.9147.
- (b) The table gives $P(Z \leq -0.84) = 0.2005$, so $P(Z > -0.84) = 1 - 0.2005 = 0.7995$.
- (c) $P(Z \leq 0.60) - P(Z \leq -1.25) = 0.7257 - 0.1056 = 0.6201$.

Answer to Exercise 19 (on page 32)

The closest entry to 0.90 in the body of the table is 0.8997, at row 1.2 and column .08. So $z \approx 1.28$.

Answer to Exercise 20 (on page 33)

- (a) Row 24, column C = 95%: $t^* = 2.064$.
- (b) The table has no row for $df = 45$, so use $df = 40$. Column C = 99% gives $t^* = 2.704$. The exact value for $df = 45$ is 2.690, slightly smaller, which confirms that rounding df down widens the interval.

Answer to Exercise 21 (on page 34)

In row 15, $t = 2.30$ falls between 2.249 (tail probability 0.02) and 2.602 (tail probability 0.01). So $0.01 < P < 0.02$. Technology gives $P = 0.0181$, inside these bounds.

Answer to Exercise 22 (on page 35)

- (a) Row 12, column .05: 21.026.
- (b) In row 4, 9.80 falls between 9.488 (tail probability 0.05) and 11.143 (tail probability 0.025). So $0.025 < P < 0.05$. Technology gives $P = 0.0439$.



INDEX

activation function, [3](#)
automatic differentiation, [5](#)

Backpropagation, [20](#)
backpropagation, [20](#)

chain rule, [19](#)
chi-square table, [34](#)
convolutional layer, [22](#)
convolutional neural network, [22](#), [24](#)
cross-entropy, [14](#)
cumulative probability, [29](#)

dead neuron, [7](#)
deep, [19](#)
deep learning, [19](#)

early stopping, [8](#)

filter, [22](#)
filter (convolution), [22](#)

logit, [13](#)
logits, [13](#)

neural network
 classification, [13](#)
 regression, [3](#)
neuron, [3](#)

PyTorch, [3](#)

ReLU, [4](#)

residual connections, [22](#)

softmax, [13](#)
softmax regression, [16](#)
standard normal table, [29](#)
 rules, [29](#)

t table, [32](#)
 reading, [32](#)
tensor, [5](#)

vanishing gradient, [21](#)