Neural Networks in R

A crash course: from a straight line to a network that beats it

Zachary Hoylman
Montana Climate Office, University of Montana
zachary.hoylman@umontana.edu

November 10, 2026

Table of contents

Before class: setup 1
How to use this document 2
Workshop roadmap 3
1 Why neural networks, and why now 3
1.1 The one diagram that matters 4
1.2 The LLM argument 5
2 A linear model is a tiny neural network 5
2.1 Why standardize? 9
3 Add a hidden layer 11
3.1 The hidden layer 12
3.2 What the hidden units are doing 16
3.3 Hyperparameters 17
4 Train, validation, test 20
4.1 Play with it: five experiments 24
4.2 Before you go: can you trust it? 28
5 k-fold cross-validation: a fair fight on real data (optional) 29
6 Beyond one hidden layer: time and space (optional) 34
6.1 Recurrent networks: memory for time series 34
6.2 Convolutional networks: neighborhoods for maps and images 36
6.3 Tools for real projects 39
Appendix: reading a model you did not train 39
Resources 39

Before class: setup

You have two files: this document (`neural_networks_in_R.qmd`) and `setup.R`. That is everything. Do these steps before class, not during. They take about ten minutes, most of it waiting.

1. **Install R (4.3 or newer) and RStudio.** On Windows, open a terminal and run:

```
winget install RProject.R Posit.RStudio
```

Or download both from https://posit.co/download/rstudio-desktop/. You do **not** need Rtools or Python.

2. **Put both files in one folder** that is not inside OneDrive, for example `C:\Users\you\neural-networks`. OneDrive-synced folders make R package installs very slow on Windows.
3. **Open `setup.R` in RStudio and click Source** (top right of the editor). It installs the packages, then downloads the engine that does the math for neural networks (about 200 MB, so use good Wi-Fi). It finishes by training a tiny network and telling you how fast your laptop is. Three `OK`s means you are ready.
4. **Open `neural_networks_in_R.qmd` in RStudio.** This is where you will work during class.

If step 3 fails, copy the red text and email it to me before class. If you cannot install software on your laptop at all, make a free account at posit.cloud, click **New Project > New RStudio Project**, upload both files with the **Upload** button in the Files pane, and do steps 3 and 4 there.

How to use this document

Run the chunks **in order, top to bottom**. Put your cursor in a chunk and press **Ctrl+Shift+Enter** (Cmd+Shift+Enter on a Mac), or click the green arrow at the top right of the chunk. Later chunks use objects made by earlier ones. If you restart R or something looks wrong, click inside the chunk you want and press **Ctrl+Alt+P** (Cmd+Option+P on a Mac). That runs every chunk above it.

Neural networks start from random numbers. If you run a chunk twice, the results will wiggle a little. That is normal, and it is part of the lesson.

Everything here runs on your laptop's CPU, and every network is deliberately tiny, a few dozen to a few thousand weights. Most chunks finish in a second or two, and the slowest takes several seconds on an older laptop. Large models train on GPUs for days, but the loop, the piles and the hyperparameters are exactly the ones you will use today.

```
# ---- Packages ----------------------------------------------------------
library(torch)     # neural networks: tensors, layers, optimizers, training
library(dplyr)     # data wrangling: filter(), mutate(), summarise()
library(ggplot2)   # plots

# ---- Random seeds ------------------------------------------------------
# Neural networks start from random numbers, and we also simulate data and
# shuffle rows at random. Seeds make your results match this document.
# Change them and the details shift, but the story stays the same.
set.seed(1)            # R's random numbers: simulated data and data splits
torch_manual_seed(1)   # torch's random numbers: the network's starting weights

# ---- A score we will use everywhere ------------------------------------
# Root mean squared error (RMSE): the typical size of a prediction miss, in
# the units of the thing you are predicting. Smaller is better.
#   observed:  the true values
#   predicted: the model's guesses for the same rows
rmse = function(observed, predicted) sqrt(mean((observed - predicted)^2))
```

```
# ---- Colors used in every plot below ----------------------------------------
col_train  = "#1b9e77"   # training data (green)
col_valid  = "#d95f02"   # validation data (orange)
col_test   = "#7570b3"   # test data (purple)
col_linear = "grey45"    # linear models (grey)
col_nn     = "#2c7fb8"   # neural networks (blue)
```

Workshop roadmap

This document is both the workshop and a reference for afterward. The live hour covers sections 1 to 4. Sections 5 and 6 are optional extensions to explore on your own.

Minutes	Live session	Where
0–10	Why neural networks; a linear model as a network; weights, biases, gradient descent and the training loop	Sections 1–2
10–25	Hidden layers, activation functions, a nonlinear species niche, model complexity	Section 3
25–45	Training, validation and test; overfitting; early stopping; data leakage	Section 4
45–55	The Shiny playground: five short experiments	"Play with it" in section 4
55–60	Can you trust it? Key takeaways	"Before you go" in section 4

During the workshop: run each chunk in sections 1 to 4 as we reach it, and do the five playground experiments. "Why standardize?" in section 2 can be skimmed live. Leave the **Try it** boxes for later.

After the workshop: the Try it boxes, section 5 (k-fold cross-validation on real ozone data) and section 6 (recurrent networks for time series and convolutional networks for maps).

Where this is heading: the checklist at the end of the document has six questions to ask of any neural network you did not train yourself, including one a chatbot writes for you. By the end of the hour you should be able to answer them.

1 Why neural networks, and why now

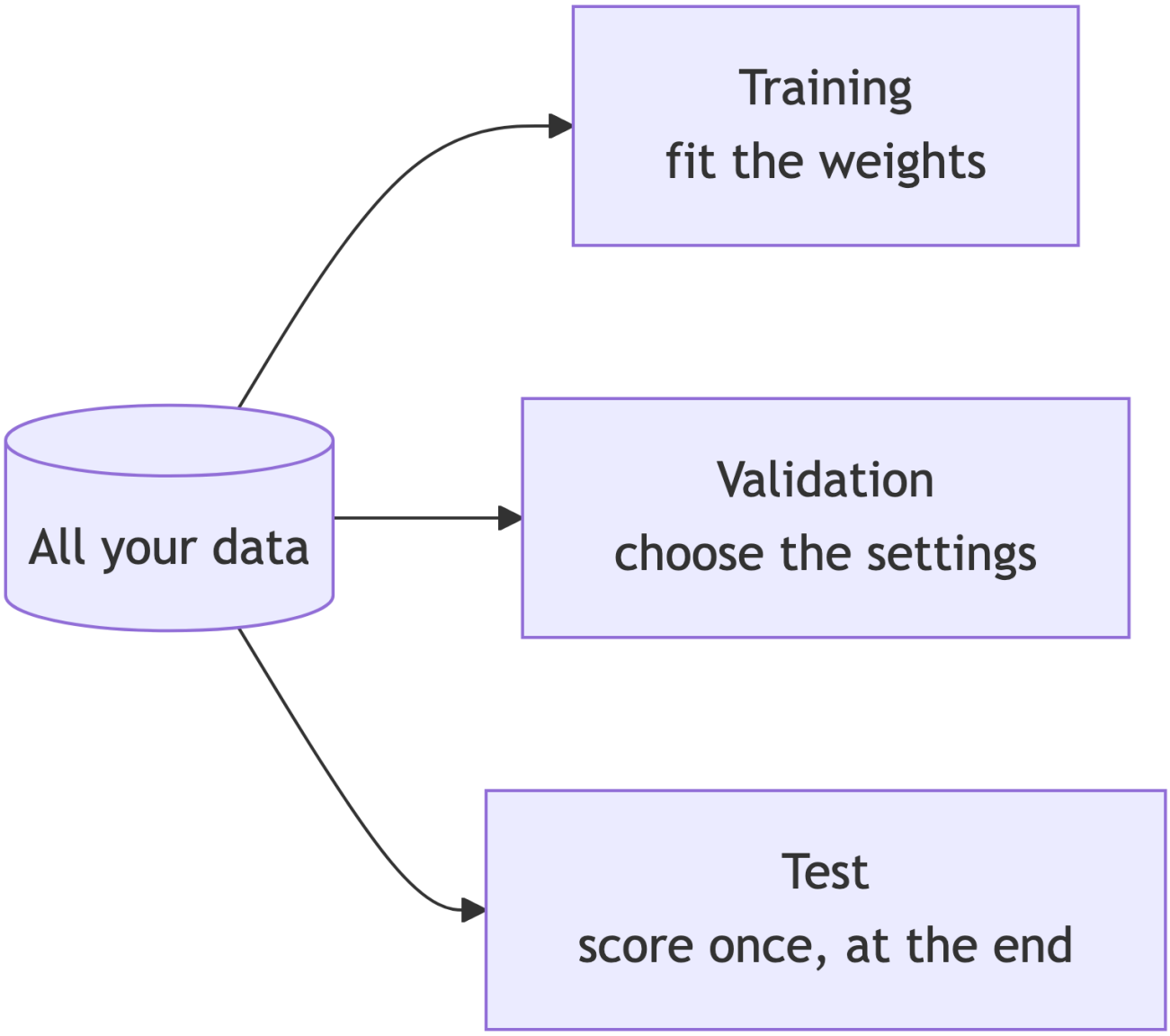
A neural network is a function with a lot of adjustable numbers in it. You show it examples, measure how wrong it is, and nudge every number a little in the direction that makes it less wrong. Repeat a few thousand times. That is the whole idea.

What makes neural networks useful is that you do not tell them the shape of the relationship. A linear model needs you to say "straight line," or "quadratic in temperature," or "interaction between wind and sun." A network finds bends and interactions on its own. In ecology that shows up in species distribution models, classifying land cover from satellite images, filling gaps in sensor records, and emulating slow process models.

What makes them dangerous is the same thing. A function flexible enough to find any shape is flexible enough to memorize noise. Most of this hour is about how to tell the difference: when a model is learning a real relationship, when it is learning noise, and whether its predictions can be trusted.

1.1 The one diagram that matters

Every honest model evaluation splits the data into three piles, each with one job.



The training pile teaches the model. The validation pile is how you choose between versions of the model. The test pile is locked in a drawer until you have made every decision, then used exactly once to report how good the model is. Every concept today, from epochs to k-fold cross-validation, is about keeping those three piles honest.

⚠ Random piles are a teaching shortcut

Today we deal rows into the three piles at random. That is fine for learning the ideas, and for our simulated survey, where every plot is independent of the others. It is often not fine for real environmental observations. Nearby sites, repeat visits to the same site and neighboring days resemble each other. A random split puts near-copies of the evaluation rows into the training pile, and the model looks better than it will be in use.

The evaluation has to match the job the model will do:

- **Predicting new locations?** Hold out whole spatial blocks.
- **Predicting future conditions?** Train on earlier years and evaluate on later ones.
- **Predicting new sites or watersheds?** Hold out entire sites or basins.

Section 5 returns to this in more detail.

1.2 The LLM argument

Ask a chatbot for a neural network on your data and you will get one in thirty seconds. It will probably run. It may even look impressive.

Running is not the same as being right. Generated code can produce convincing numbers while relying on a weak baseline, a validation design that does not match the science, or information quietly leaking from the test data into training. Those mistakes make a model look better on paper than it will be in the field, and they are easy to make and hard to spot.

AI tools can often catch these problems when you ask, especially when you give them enough context. But whether a validation strategy is scientifically appropriate depends on things the code does not show: how the data were sampled, how observations depend on each other in space and time, and what the model will be used for. That judgment is yours.

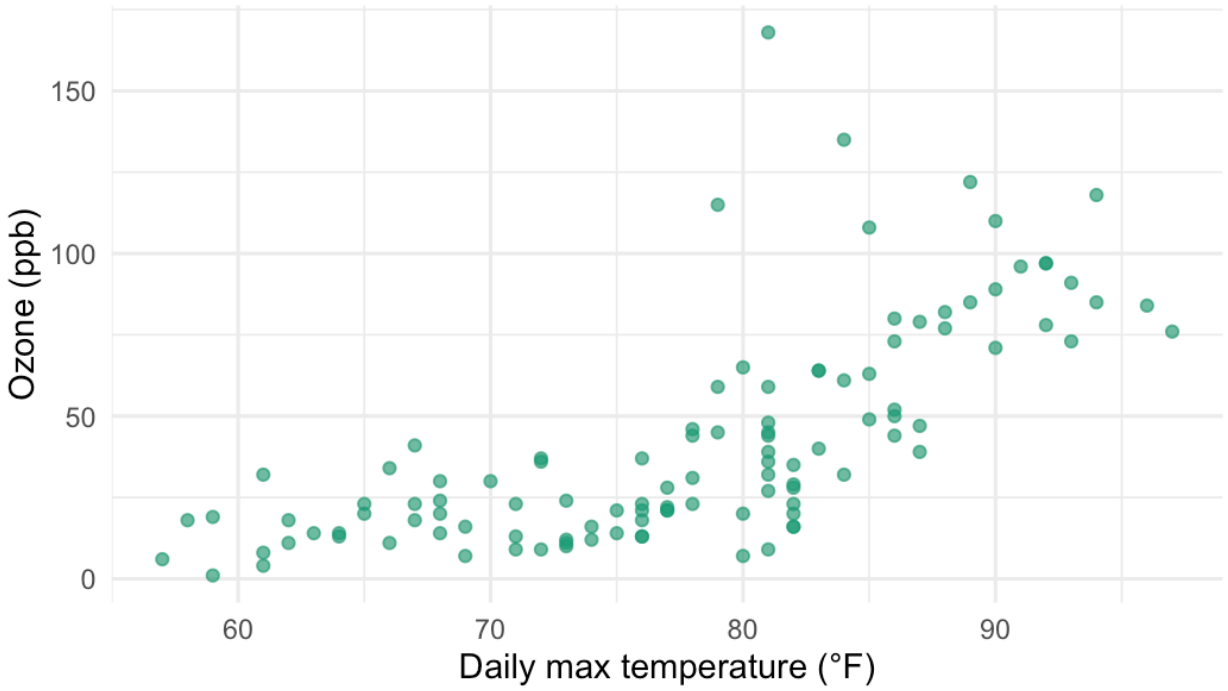
If you understand the three piles, you can read generated code, supply that context and ask the right questions. Each section ends with one you can take to a model yourself.

2 A linear model is a tiny neural network

Start with something you already know. The `airquality` data that ships with R has daily ozone, temperature, wind and sunlight in New York in 1973. Hot days have more ozone.

```
# airquality ships with R: 153 days of New York air quality in 1973.
# na.omit() drops days with any missing measurement, leaving 111 days.
aq = na.omit(airquality)

# One point per day: ozone against that day's max temperature
ggplot(aq, aes(x = Temp, y = Ozone)) +
  geom_point(color = col_train, alpha = 0.7) +
  labs(x = "Daily max temperature (°F)", y = "Ozone (ppb)") +
  theme_minimal(base_size = 14)
```



Fit a straight line with `lm()`. One change from usual: we **standardize** temperature first, subtracting its mean and dividing by its standard deviation. The result, sometimes called a z-score, says how many standard deviations a day is above or below an average day. It does not change the line, only the units of the slope. Why it matters gets its own subsection below.

```
# Standardize: subtract the mean, then divide by the standard deviation.
# An average day (77.9 °F) becomes 0. A day one standard deviation warmer
# (about 87 °F) becomes 1. Most days land between -2 and 2.
aq$temp_z = (aq$Temp - mean(aq$Temp)) / sd(aq$Temp)

# The ordinary linear model. lm() solves for the best line exactly, in one step.
fit_lm = lm(Ozone ~ temp_z, data = aq)

# Intercept: ozone on an average-temperature day.
# Slope: extra ppb of ozone for each standard deviation of temperature.
coef(fit_lm)
```

```
(Intercept)      temp_z
   42.09910    23.24464
```

Now fit the *same model* as a neural network. A network with no hidden layer is one weight times the input, plus one bias: `y = w * x + b`. That is a straight line. The difference is how it is fit. `lm()` solves for the answer directly. A network starts from random numbers and walks downhill.

```
# ---- The data, in torch's format ----------------------------------------
# torch works on "tensors": arrays it can do calculus on. A layer expects a
# matrix with one row per observation and one column per input.
x = torch_tensor(matrix(aq$temp_z))              # inputs: 111 days x 1 column

# The answers. as.numeric() because torch needs decimals, and Ozone is
# stored as whole numbers.
y = torch_tensor(matrix(as.numeric(aq$Ozone)))

# ---- The model ----------------------------------------------------------
# y = w * x + b: one weight w (the slope) and one bias b (the intercept).
# Both start as random numbers. The seed makes them the same every run.
torch_manual_seed(1)
net = nn_linear(in_features = 1, out_features = 1)

# ---- The optimizer ------------------------------------------------------
# The optimizer does the walking downhill. SGD is stochastic gradient descent:
# every step, each number moves by (learning rate x its gradient).
optimizer = optim_sgd(net$parameters, lr = 0.1)   # lr = learning rate: the step size

# ---- The training loop --------------------------------------------------
losses = numeric(100)            # somewhere to record the loss after each epoch
for (epoch in 1:100) {           # 100 epochs = 100 passes through all 111 days

  # 1. Forget the last step's gradients. torch adds them up otherwise.
  optimizer$zero_grad()

  # 2. Forward pass: predict ozone for every day.
  pred = net(x)

  # 3. Loss: mean squared error between the predictions and the answers.
  loss = nnf_mse_loss(pred, y)

  # 4. Backward pass: which way should each number move, and how steeply?
  loss$backward()

  # 5. Move each number a small step downhill.
  optimizer$step()

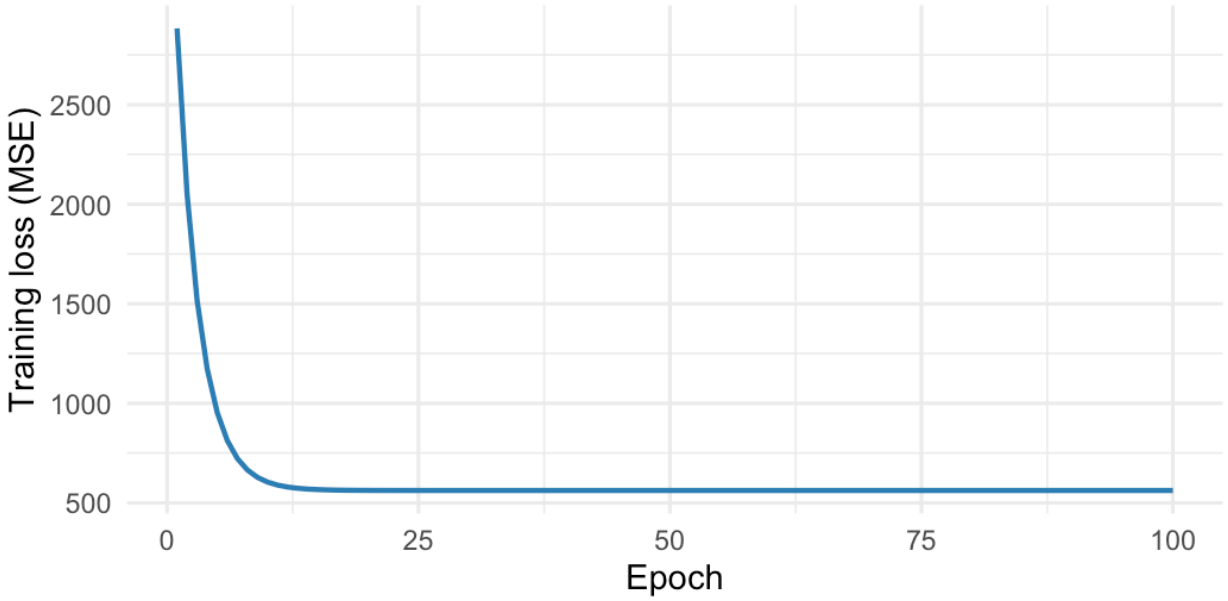
  losses[epoch] = loss$item()    # $item() pulls one plain number out of a tensor
}

# ---- Did it find the same line as lm()? ---------------------------------
c(network_slope     = net$weight$item(), lm_slope     = unname(coef(fit_lm)[2]),
  network_intercept = net$bias$item(),   lm_intercept = unname(coef(fit_lm)[1]))
```

```
    network_slope          lm_slope network_intercept      lm_intercept
          23.24464          23.24464          42.09909          42.09910
```

Same answer. Here is the walk downhill:

```
# The loss after each of the 100 epochs: high at the random start, then down
tibble(epoch = 1:100, loss = losses) |>
  ggplot(aes(x = epoch, y = loss)) +
  geom_line(color = col_nn, linewidth = 1) +
  labs(x = "Epoch", y = "Training loss (MSE)") +
  theme_minimal(base_size = 14)
```



Those five lines inside the loop are every neural network you will ever train, from this one to the models behind chatbots:

1. **Zero the gradients** left over from the last step.
2. **Make predictions** with the current weights. This is the forward pass.
3. **Calculate the loss**: how wrong those predictions are.
4. **Backpropagate**: work out which way each weight should move to shrink the loss.
5. **Update the weights**: move each one a small step that way.

The vocabulary:

- **Weights and biases:** the adjustable numbers. Here there are two. Large models have billions.
- **Loss:** one number that says how wrong the model is. Mean squared error for predicting amounts.
- **Gradient:** for each weight, which direction makes the loss go down, and how steeply. `loss$backward()` computes all of them at once.
- **Learning rate:** how big a step to take downhill. Our first **hyperparameter**: a setting you choose, not one the model learns.

- **Epoch:** one pass through all the training data.

> **Try it**
>
> Change the learning rate to `0.01`, then `0.5`, then `1`, rerunning the loop each time. What happens to the loss curve and to the final slope? Which one would you call "best," and would you expect the same value to be best on a different problem?

> **Ask the model**
>
> Paste the training loop into a chatbot and ask: "Explain what each of the five lines inside this loop does. What would go wrong if I deleted `optimizer$zero_grad()`?" Then delete it and see if the model was right.

2.1 Why standardize?

Here is the same loop on raw temperatures, 57 to 97 °F, instead of standardized ones. First with the step size that just worked, then with a step 1,000 times smaller, then with that small step for 50 times as many epochs.

```
# The loop above, wrapped in a function so we can try different settings.
#   x_input: the input tensor to train on
#   lr:      learning rate
#   epochs:  number of passes through the data
# Returns one row: the final loss, slope and intercept.
gd_line = function(x_input, lr, epochs = 100) {
  torch_manual_seed(1)                              # same random start as before
  net = nn_linear(1, 1)                             # one weight, one bias
  optimizer = optim_sgd(net$parameters, lr = lr)
  for (epoch in 1:epochs) {
    optimizer$zero_grad()
    loss = nnf_mse_loss(net(x_input), y)
    loss$backward()
    optimizer$step()
  }
  tibble(learning_rate = lr, epochs = epochs, final_loss = loss$item(),
         slope = net$weight$item(), intercept = net$bias$item())
}

# Raw temperatures in °F, NOT standardized
x_raw = torch_tensor(matrix(as.numeric(aq$Temp)))

bind_rows(
  gd_line(x_raw, lr = 0.1),                     # the step size that worked above
  gd_line(x_raw, lr = 0.0001),                  # a step 1,000 times smaller
```

```
  gd_line(x_raw, lr = 0.0001, epochs = 5000)    # the small step, 50 times longer
)
```

```
# A tibble: 3 × 5
  learning_rate epochs final_loss   slope intercept
          <dbl>  <dbl>      <dbl>   <dbl>     <dbl>
1         0.1      100       NaN  NaN       NaN
2         0.0001   100       879.   0.575    -0.484
3         0.0001  5000       870.   0.602    -2.58
```

```
# For comparison, the right answer in raw units, from lm():
coef(lm(Ozone ~ Temp, data = aq))
```

```
(Intercept)        Temp
 -147.64607     2.43911
```

With the step size that worked before, the loss explodes and every number turns into `NaN`, "not a number." Shrink the step 1,000 times and it stops exploding, but it barely moves. Even after 5,000 epochs the line it has found runs close to the origin, with a slope of about 0.6 and an intercept near zero. The right answer is a slope of 2.4 and an intercept of −148.

The reason is the shape of the problem. Picture the loss as a landscape over the two numbers the model is learning, the slope and the intercept. Every gradient for the slope gets multiplied by the input values, so with temperatures around 80 the landscape is a long, narrow canyon. It is extremely steep across the slope direction and nearly flat along the intercept direction. One step size has to serve both. A step big enough to travel along the flat floor launches you off the steep walls, which is the explosion. A step small enough to stay on the walls crawls along the floor and would need millions of epochs. Standardizing turns the canyon into a round bowl, where one step size suits every direction. Same data, same line, different units.

Networks care about this even more than straight lines do:

- **Several inputs, several units.** In section 5 sunlight runs from 7 to 334, wind from 2 to 21 and temperature from 57 to 97. Unstandardized, the input with the biggest numbers dominates every step, whether or not it matters most.
- **Starting weights.** torch starts every weight and bias at a small random number that assumes inputs between about −2 and 2. That places each hidden unit's kink somewhere in that range. Feed it 57 to 97 and every kink sits far from the data, so units start out stuck all the way on or all the way off.
- **Weight decay** pulls every weight toward zero by the same rule. That is only fair if every input is on the same scale.
- **The output, too.** We also standardize the thing we predict, such as abundance or ozone. A new network's guesses sit near 0 with a spread of about 1, so standardized answers start it in the right ballpark. Predictions are converted back with the same mean and standard deviation.

One rule, which section 4 explains: the mean and standard deviation come from the **training data only**, and those same two numbers are then applied to everything else.

3 Add a hidden layer

A straight line is a fine model until the world is not straight. Species have niches: too cold and too hot are both bad. Here is a made-up species, most abundant near 14 °C. We measure it with a continuous **abundance index**, such as a density score from a survey, rather than a raw count. Because we made the species up, we know the true curve, which lets us see exactly what each model gets right and wrong.

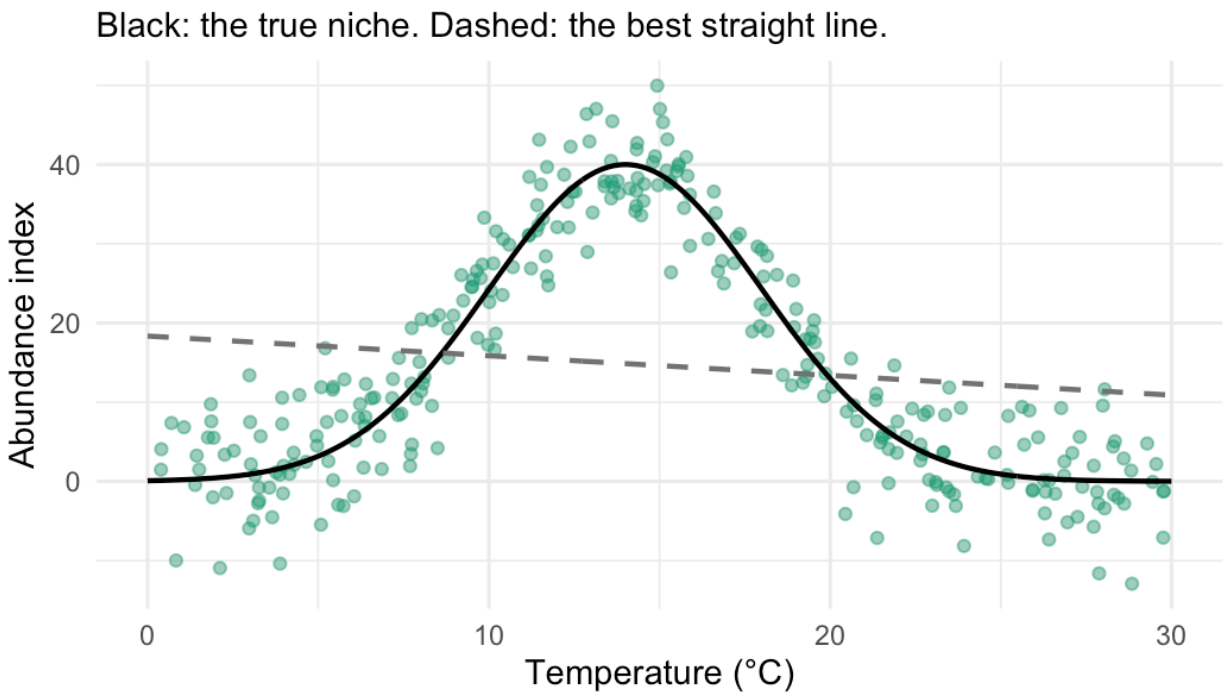
```
# ---- A made-up species with a temperature niche ----------------------------
# The true relationship: a bell curve that peaks at an index of 40 at 14 °C,
# with a niche width (standard deviation) of 4 °C.
true_curve = function(temp) 40 * exp(-((temp - 14)^2) / (2 * 4^2))

# 300 survey plots at random temperatures between 0 and 30 °C. Each reading is
# the true index plus random measurement noise (sd = 5).
niche = tibble(temp = runif(300, 0, 30)) |>
  mutate(abundance = true_curve(temp) + rnorm(n(), 0, 5))

# An even grid of temperatures, used only to draw smooth curves
grid = tibble(temp = seq(0, 30, length.out = 300))

# The best possible straight line through the data
fit_line = lm(abundance ~ temp, data = niche)

# Points: the data. Black: the truth. Dashed: the straight line.
ggplot(niche, aes(x = temp, y = abundance)) +
  geom_point(color = col_train, alpha = 0.5) +
  geom_line(data = grid, aes(y = true_curve(temp)), linewidth = 1) +
  geom_line(data = grid, aes(y = predict(fit_line, grid)),
            color = col_linear, linewidth = 1, linetype = "dashed") +
  labs(x = "Temperature (°C)", y = "Abundance index",
       subtitle = "Black: the true niche. Dashed: the best straight line.") +
  theme_minimal(base_size = 14)
```



The straight line is not wrong because of bad fitting. It is the best straight line there is. The model is too simple for the question.

Some readings fall below zero. That is the measurement noise: near the edges of the niche the true index is close to zero, and noise pushes some readings below it. We keep it because it makes the noise easy to see. Real counts cannot be negative. For counts you would use a model built for them, such as a Poisson or negative binomial likelihood, or a network whose output cannot go below zero.

3.1 The hidden layer

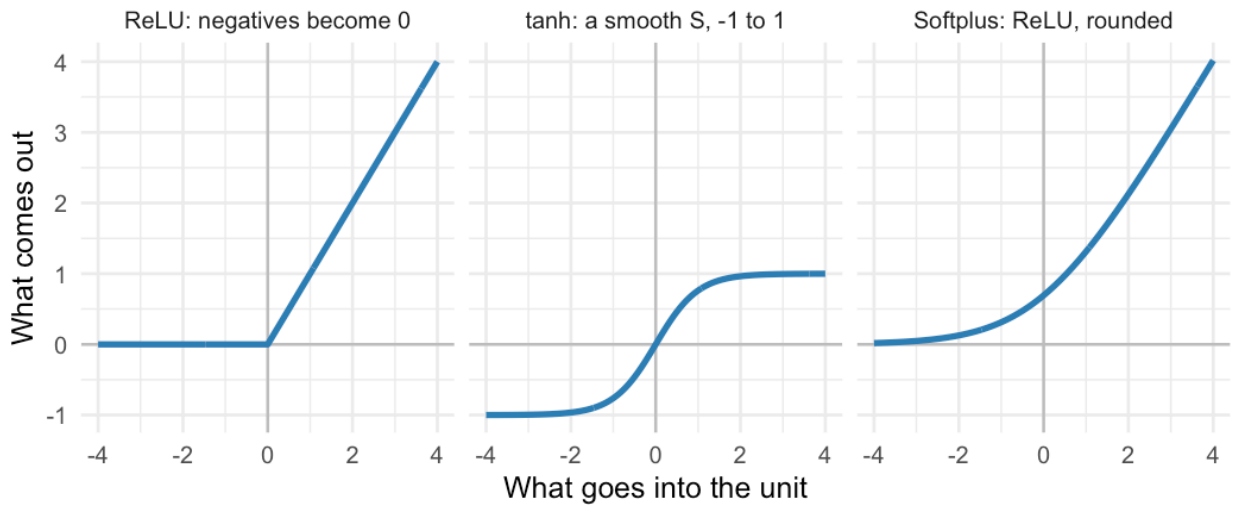
A **hidden layer** sits between input and output. Each **hidden unit** takes the input, applies its own weight and bias, then bends the result with an **activation function**. We use ReLU: negative numbers become zero, positive numbers pass through. Every unit is a line with a kink in it. The output layer adds the units together, each with its own weight. Add up enough kinked lines and you can draw almost any curve.

ReLU is not the only bend. Here it is next to two smooth alternatives. **tanh** squashes any input into a smooth S-curve between −1 and 1, and it was the standard choice before ReLU took over. **Softplus** is ReLU with the kink rounded off. A network built from smooth bends draws smooth curves, and one built from ReLU draws curves out of straight pieces.

```
# An activation function is a plain function: one number in, one number out.
# These are torch's own versions, the same ones the network uses.
z = seq(-4, 4, length.out = 401)   # a range of inputs to feed in
bends = bind_rows(
  tibble(z = z, out = as.numeric(nnf_relu(torch_tensor(z))),
         name = "ReLU: negatives become 0"),
  tibble(z = z, out = as.numeric(torch_tanh(torch_tensor(z))),
         name = "tanh: a smooth S, -1 to 1"),
```

```
  tibble(z = z, out = as.numeric(nnf_softplus(torch_tensor(z))),
         name = "Softplus: ReLU, rounded")
) |>
  mutate(name = factor(name, levels = unique(name)))   # keep this panel order

ggplot(bends, aes(x = z, y = out)) +
  geom_hline(yintercept = 0, color = "grey75") +      # axes through zero
  geom_vline(xintercept = 0, color = "grey75") +
  geom_line(color = col_nn, linewidth = 1.2) +
  facet_wrap(~ name) +
  labs(x = "What goes into the unit", y = "What comes out") +
  theme_minimal(base_size = 12)
```



This function trains a network with one hidden layer. It is the same five-line loop as before, wrapped up so we can reuse it. The activation is one of its settings, ReLU unless you choose another. It also records the loss on validation data, if we give it some. We will need that in the next section.

```
# Train a neural network with one hidden layer.
#   x, y:          training inputs and answers, as tensors (one row per observation)
#   hidden:        number of hidden units, the number of kinked lines
#   epochs:        number of passes through the training data
#   lr:            learning rate, the step size
#   decay:         weight decay, how hard to pull weights toward zero
#   activation:    the bend: "relu", "tanh" or "softplus"
#   x_val, y_val:  optional validation data. Only used to WATCH the loss,
#                  never to change the weights.
# Returns a list: the trained network, and a table of the loss after every epoch.
train_nn = function(x, y, hidden = 16, epochs = 300, lr = 0.03, decay = 0,
                    activation = "relu", x_val = NULL, y_val = NULL) {
```

```
  # ---- The activation: pick torch's layer by name --------------------------
  bend = switch(activation,
    relu     = nn_relu,       # kinked: negatives become 0
    tanh     = nn_tanh,       # smooth S-curve between -1 and 1
    softplus = nn_softplus    # ReLU with the kink rounded off
  )

  # ---- The architecture: inputs -> hidden layer -> one output ---------------
  net = nn_sequential(
    nn_linear(ncol(x), hidden),   # inputs -> hidden units
    bend(),                       # the bend (ReLU unless you choose another)
    nn_linear(hidden, 1)          # hidden units -> one output
  )

  # ---- The optimizer -------------------------------------------------------
  # Adam is gradient descent that adapts the step size for each weight as it
  # goes, so it is much less fussy about the learning rate than plain SGD.
  # weight_decay gently pulls every weight toward zero on every step.
  optimizer = optim_adam(net$parameters, lr = lr, weight_decay = decay)

  # ---- The same five-line loop as section 2, plus bookkeeping ---------------
  train_loss = numeric(epochs)          # training loss after each epoch
  valid_loss = rep(NA_real_, epochs)    # validation loss (stays NA if none given)
  for (epoch in 1:epochs) {
    optimizer$zero_grad()               # 1. forget old gradients
    loss = nnf_mse_loss(net(x), y)      # 2 and 3. predict, then score
    loss$backward()                     # 4. compute gradients
    optimizer$step()                    # 5. step downhill
    train_loss[epoch] = loss$item()

    # Score the validation data too, if we were given some. with_no_grad()
    # means "just predict, do not track gradients", so the validation data
    # can never nudge the weights.
    if (!is.null(x_val)) {
      valid_loss[epoch] = with_no_grad(nnf_mse_loss(net(x_val), y_val))$item()
    }
  }

  list(net = net,
       history = tibble(epoch = 1:epochs, train = train_loss, validation = valid_loss))
}
```

Standardize the input and the output, train, then convert predictions back to abundance units. For now we use all the data for training. That is cheating, and the next section fixes it.

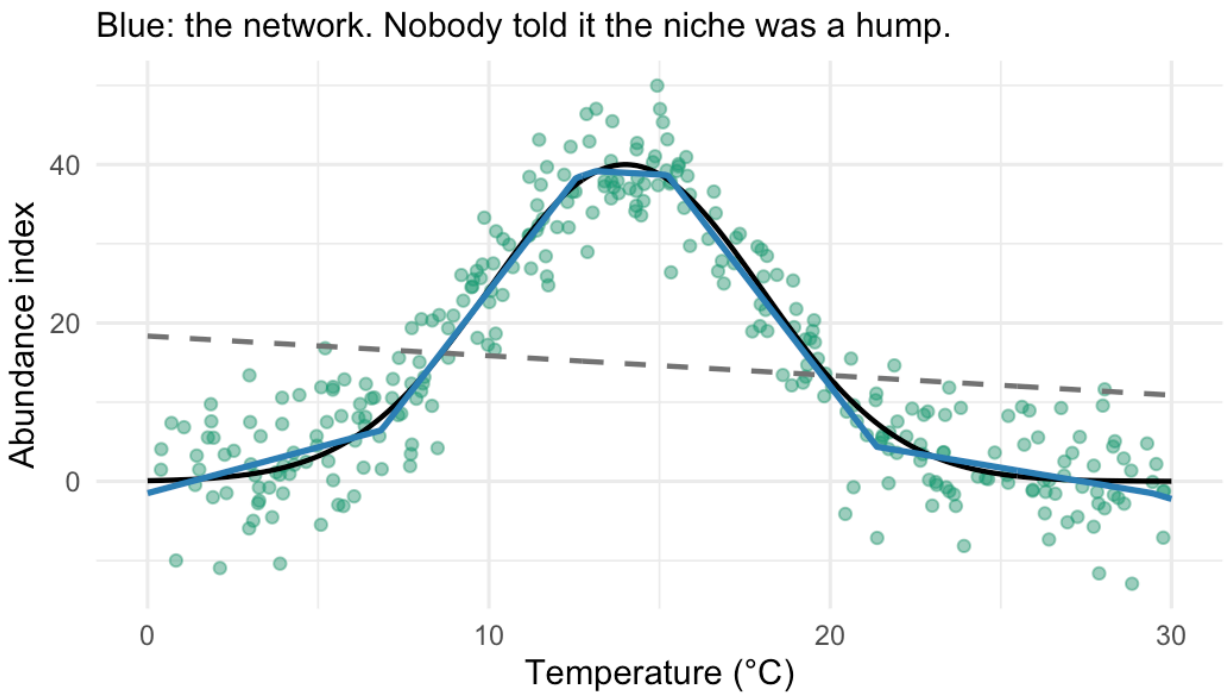
```
# ---- Standardizing numbers (section 2 explains why) -------------------------
# Remember the mean and standard deviation of the input and of the output...
x_mean = mean(niche$temp);      x_sd = sd(niche$temp)
y_mean = mean(niche$abundance); y_sd = sd(niche$abundance)

# ...and write three small helpers so we never standardize by hand:
# °C -> standardized input tensor
to_x   = function(temp) torch_tensor(matrix((temp - x_mean) / x_sd))
# index -> standardized answer tensor
to_y   = function(abundance) torch_tensor(matrix((abundance - y_mean) / y_sd))
# network output -> back to the index
from_y = function(tensor) as.numeric(tensor) * y_sd + y_mean

# Predict abundance at any temperatures with a trained network.
predict_nn = function(net, temp) {
  force(net)   # R only evaluates `net` when it is first used. force() does it
               # here, before gradients are switched off, in case `net` still
               # needs training. That matters in the Shiny app later.
  # standardize, predict, convert back to the index
  from_y(with_no_grad(net(to_x(temp))))
}

# ---- Train: 16 hidden units for 300 epochs (train_nn's defaults) ------------
torch_manual_seed(1)
fit = train_nn(to_x(niche$temp), to_y(niche$abundance), hidden = 16)

# The same plot as before, with the network added in blue
ggplot(niche, aes(x = temp, y = abundance)) +
  geom_point(color = col_train, alpha = 0.5) +
  geom_line(data = grid, aes(y = true_curve(temp)), linewidth = 1) +
  geom_line(data = grid, aes(y = predict(fit_line, grid)),
            color = col_linear, linewidth = 1, linetype = "dashed") +
  geom_line(data = grid, aes(y = predict_nn(fit$net, temp)),
            color = col_nn, linewidth = 1.2) +
  labs(x = "Temperature (°C)", y = "Abundance index",
       subtitle = "Blue: the network. Nobody told it the niche was a hump.") +
  theme_minimal(base_size = 14)
```



3.2 What the hidden units are doing

Here are the 16 hidden units pulled apart. Each faint line is one unit's kinked line times its output weight. The blue curve is their sum, shifted up or down by one constant (the output bias). No single unit knows anything about niches. Some are flat at zero across this whole range, switched off. Others are large and mostly cancel each other out. Only the sum means anything.

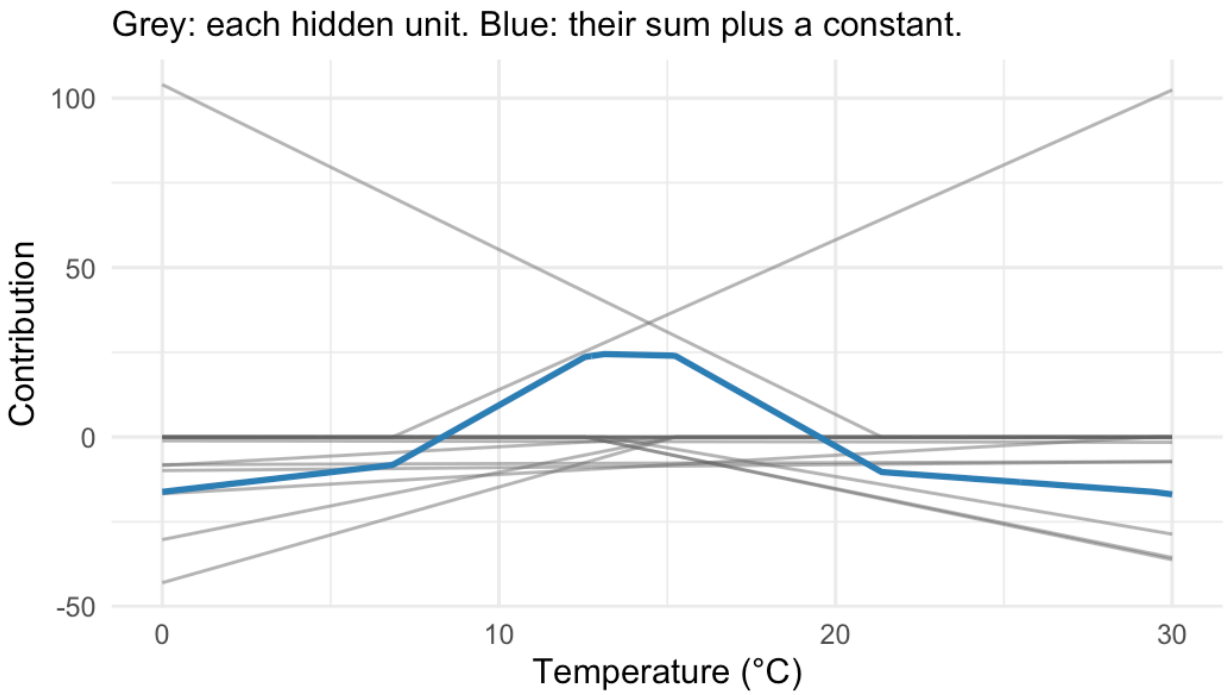
```
# A network built with nn_sequential() is a list of layers:
#   fit$net[[1]] = the first linear layer, fit$net[[2]] = the ReLU,
#   fit$net[[3]] = the output layer.
# Run the grid through the first two layers only, to see what each hidden
# unit outputs after its bend: a table of 300 temperatures x 16 units.
hidden_out = with_no_grad(fit$net[[2]](fit$net[[1]](to_x(grid$temp))))

# The output layer's weight for each hidden unit: how much that unit counts
out_weight = as.numeric(fit$net[[3]]$weight)                  # 16 numbers

# Each unit's contribution = its output x its weight, in abundance units
pieces = as_array(hidden_out) *
  matrix(out_weight, nrow(grid), length(out_weight), byrow = TRUE) * y_sd

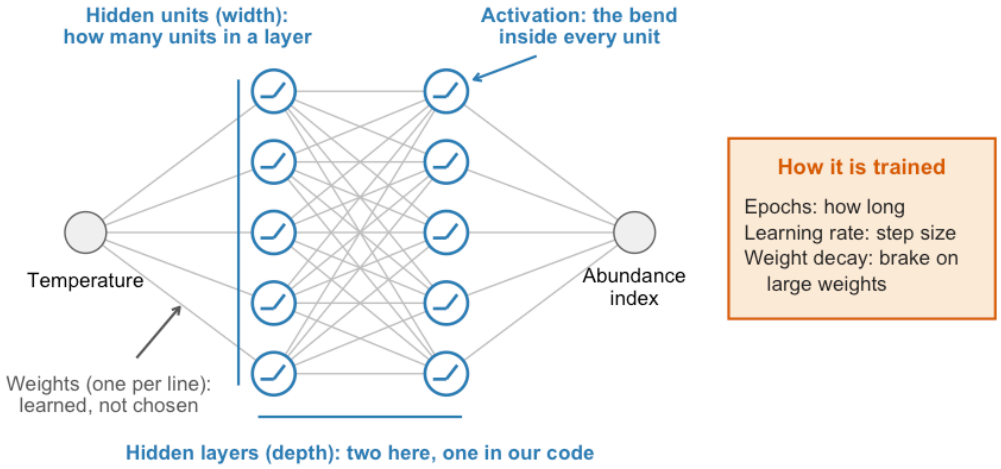
# Reshape to one row per temperature per unit, then draw each unit as a line
tibble(temp = rep(grid$temp, ncol(pieces)),
       unit = rep(seq_len(ncol(pieces)), each = nrow(grid)),
       contribution = as.vector(pieces)) |>
  ggplot(aes(x = temp, y = contribution, group = unit)) +
```

```
  geom_line(alpha = 0.5, color = "grey40") +
  geom_line(data = grid, aes(x = temp, y = predict_nn(fit$net, temp) - y_mean),
            inherit.aes = FALSE, color = col_nn, linewidth = 1.2) +
  labs(x = "Temperature (°C)", y = "Contribution",
       subtitle = "Grey: each hidden unit. Blue: their sum plus a constant.") +
  theme_minimal(base_size = 14)
```



3.3 Hyperparameters

The settings you choose, rather than the model learning, are **hyperparameters**. Some decide the shape of the network, and some decide how it is trained:



In blue, the shape of the network: how many units, how many layers, and the bend inside each unit. In orange, how it is trained. The lines between units are the weights. They are not hyperparameters: they are what training learns. Here is what each setting does when it is too small or too big:

Hyperparameter	What it controls	Too small	Too big
Hidden units	How many kinked lines	Too stiff to fit the shape	Enough freedom to memorize noise
Hidden layers	Kinks of kinks	Can miss complex interactions	Harder to train, more to tune
Epochs	How long it trains	Stops before it fits	Keeps going into the noise
Learning rate	Step size downhill	Painfully slow	Bounces around or blows up
Weight decay	Pull toward small weights	No brake on wiggles	Too smooth, underfits

One more setting is not a size at all: the **activation**, the shape of each unit's bend.

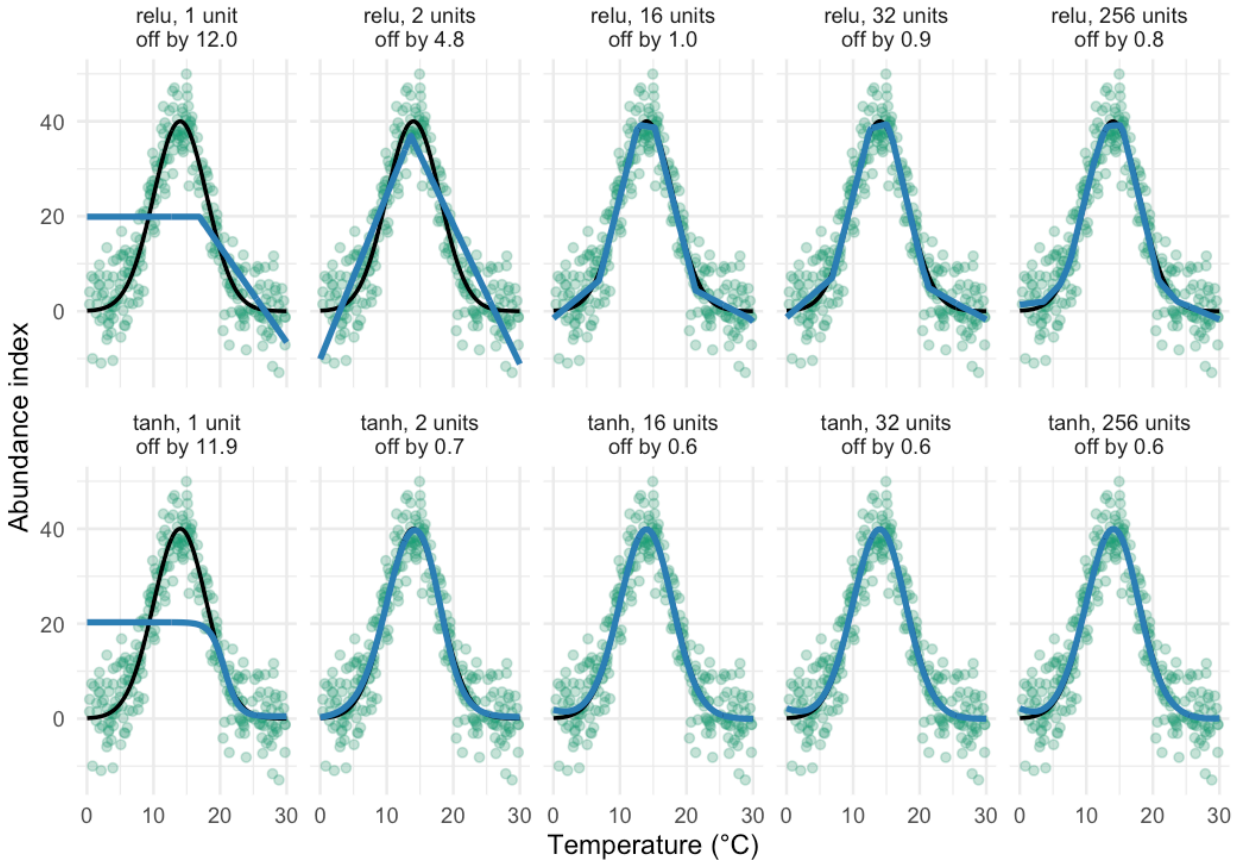
Here are the first row and the activation in action. The two lines at the top of the chunk set which widths and which bends to compare. Change them and rerun to try any combination you like.

```
# ---- Choose what to compare ------------------------------------------------
# Edit these two lines and rerun the chunk.
#   hidden_sizes: any widths, for example c(1, 2, 4, 8, 16, 64)
#   activations:  any of "relu", "tanh", "softplus"
# Every combination gets its own panel. Each takes about a second on a slow
# laptop, so keep it to a dozen or so.
# (Coming back here after section 4? The numbers will shift slightly, because
# section 4 changes the standardizing numbers. The pattern stays the same.)
hidden_sizes = c(1, 2, 16, 32, 256)
activations  = c("relu", "tanh")

# ---- Train one network per combination -------------------------------------
# Every network starts from the same seed, so adding a setting does not
# change the others. Each one's curve is recorded across the temperature grid.
settings = expand.grid(hidden = hidden_sizes, activation = activations,
                       stringsAsFactors = FALSE)
fits = lapply(seq_len(nrow(settings)), function(i) {
  torch_manual_seed(1)
  f = train_nn(to_x(niche$temp), to_y(niche$abundance),
               hidden = settings$hidden[i], activation = settings$activation[i])
  tibble(temp = grid$temp, abundance = predict_nn(f$net, grid$temp),
         hidden = settings$hidden[i], activation = settings$activation[i])
})
fits = bind_rows(fits)   # stack them into one table
```

```
# ---- Label each panel -------------------------------------------------------
# "relu, 1 unit", "tanh, 16 units", ... plus how far that curve sits from the
# true niche on average (in index units; smaller is better)
panel_labels = fits |>
  summarise(off_by = rmse(true_curve(temp), abundance), .by = c(activation, hidden))
|>
  arrange(match(activation, activations), hidden) |>
  mutate(label = sprintf("%s, %d unit%s\noff by %.1f",
                         activation, hidden, ifelse(hidden == 1, "", "s"), off_by))
fits = fits |>
  left_join(select(panel_labels, activation, hidden, label),
            by = c("activation", "hidden")) |>
  mutate(panel = factor(label, levels = panel_labels$label))

# ---- Plot: one row per activation, one column per width ----------------------
ggplot(niche, aes(x = temp, y = abundance)) +
  geom_point(color = col_train, alpha = 0.3) +                        # the data
  geom_line(data = grid, aes(y = true_curve(temp)), linewidth = 0.8) + # the truth
  geom_line(data = fits, color = col_nn, linewidth = 1.2) +           # each network
  facet_wrap(~ panel, ncol = length(hidden_sizes)) +
  labs(x = "Temperature (°C)", y = "Abundance index") +
  theme_minimal(base_size = 11)
```



With ReLU, one unit is one kinked line. It cannot make a hump. Two units can make a tent, and it takes more to round it into a smooth curve. With tanh, two units already make the hump: one S-curve going up and one coming down. Matching the bend to the shape of the problem can matter as much as adding units.

> **Try it**
>
> Add 4 and 8 to `hidden_sizes`, and add `"softplus"` to `activations`, then rerun the chunk. Then give the network a **second hidden layer**: inside `train_nn()`, add `nn_linear(hidden, hidden), bend(),` after the first `bend()`, rerun that chunk, then rerun the `hidden_sizes` chunk. Does more always look better here? Why might this data not show the danger yet?

> **Ask the model**
>
> "Explain what a ReLU hidden unit does to its input. Then explain why a network with many layers but no activation functions between them is still just a linear model."

4 Train, validation, test

Everything so far was scored on the data it was trained on. That tells you how well a model can *memorize*, not how well it will *predict*. To see the difference we need less data and more noise, which is what real field data usually looks like.

```
set.seed(2)   # a new seed for a new, smaller survey
# Same species, but only 150 plots and noisier readings (sd = 8 instead of 5)
small = tibble(temp = runif(150, 0, 30)) |>
  mutate(abundance = true_curve(temp) + rnorm(n(), 0, 8))

# Deal the 150 rows into three piles at random: 90 train, 30 validation, 30 test.
# Random is fine here because every simulated plot is independent.
# rep() makes the 150 labels, and sample() shuffles them onto the rows.
small$pile = sample(rep(c("train", "validation", "test"), times = c(90, 30, 30)))
count(small, pile)   # check the pile sizes
```

```
# A tibble: 3 × 2
  pile           n
  <chr>      <int>
1 test          30
2 train         90
3 validation    30
```

```
train = filter(small, pile == "train")        # fits the weights
valid = filter(small, pile == "validation")   # chooses the settings
test  = filter(small, pile == "test")         # into the drawer until the very end
```

One quiet but important step: the standardizing numbers must come **from the training pile only**. If the mean and standard deviation include validation or test rows, information about those rows has leaked into training. Here the effect is small. In real projects, leakage like this is one of the most common reasons a model looks better on paper than in the field.

```
# Overwrite the standardizing numbers using the TRAINING rows only.
# to_x(), to_y(), from_y() and predict_nn() from section 3 read these four
# numbers, so from here on they are leak-free. The validation and test rows
# get standardized with the training pile's mean and sd, not their own.
x_mean = mean(train$temp);      x_sd = sd(train$temp)
y_mean = mean(train$abundance); y_sd = sd(train$abundance)
```

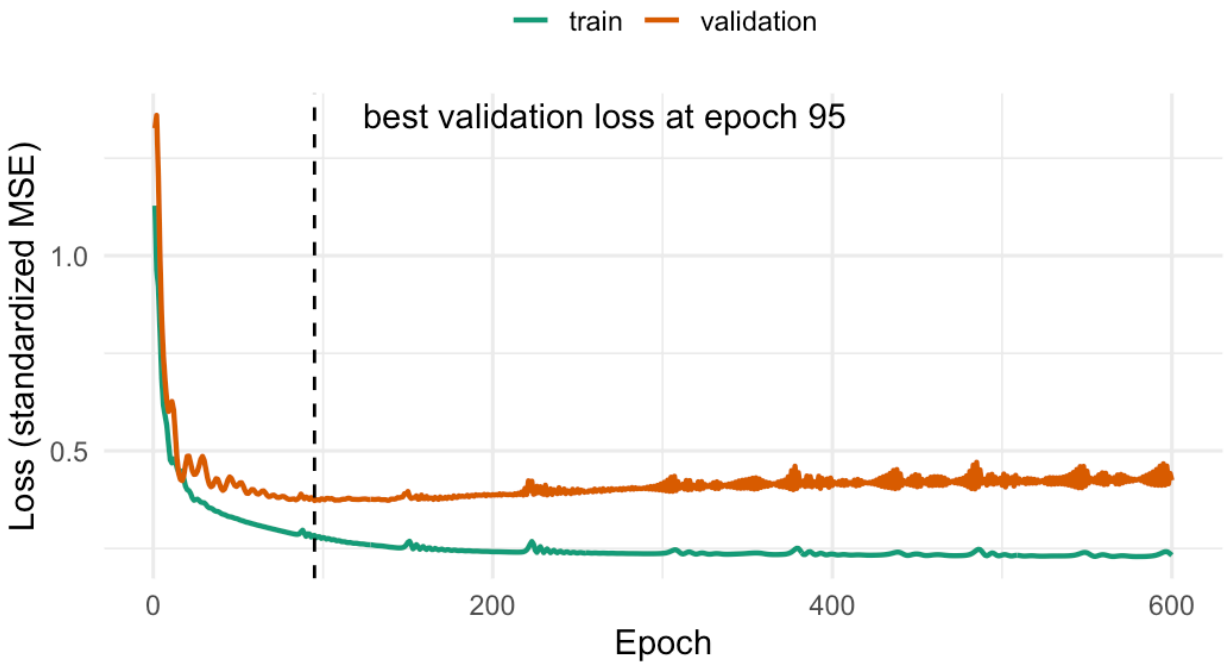
Now train a network that is too big for 90 points, 64 hidden units for 600 epochs, and watch the loss on **both** the training and validation piles as it goes.

```
# Train on the training pile, and hand over the validation pile so the
# function records its loss after every epoch. It never trains on it.
torch_manual_seed(1)
big = train_nn(to_x(train$temp), to_y(train$abundance), hidden = 64, epochs = 600,
               x_val = to_x(valid$temp), y_val = to_y(valid$abundance))

# The epoch where validation loss was lowest
```

```
best_epoch = which.min(big$history$validation)

# Both loss curves on one plot. pivot_longer() stacks the train and
# validation columns into one, so ggplot can color them by pile.
big$history |>
  tidyr::pivot_longer(c(train, validation), names_to = "pile", values_to = "loss") |>
  ggplot(aes(x = epoch, y = loss, color = pile)) +
  geom_line(linewidth = 1) +
  geom_vline(xintercept = best_epoch, linetype = "dashed") +
  annotate("text", x = best_epoch, y = max(big$history$validation), hjust = -0.1,
           label = paste("best validation loss at epoch", best_epoch)) +
  scale_color_manual(values = c(train = col_train, validation = col_valid)) +
  labs(x = "Epoch", y = "Loss (standardized MSE)", color = NULL) +
  theme_minimal(base_size = 14) +
  theme(legend.position = "top")
```



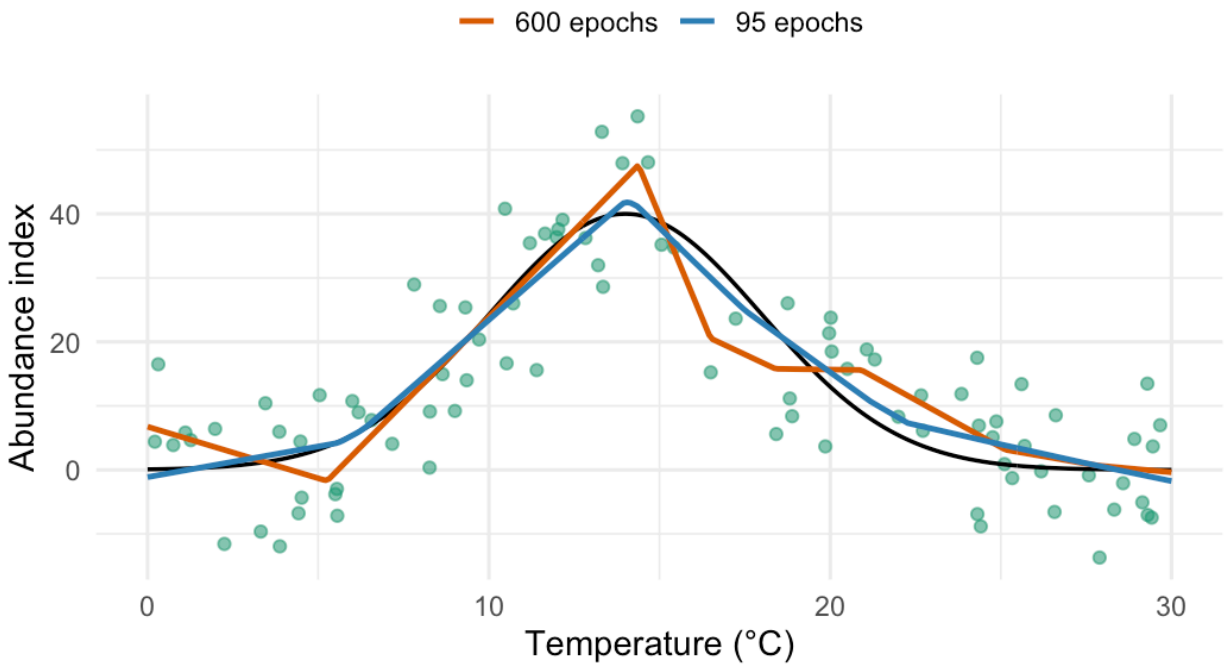
This is the most important plot in machine learning. Training loss keeps falling: the network keeps getting better at the data it has seen. Validation loss falls, bottoms out, then climbs: past that point the network is learning the noise in those 90 rows, not the niche. That is **overfitting**, and you can only see it because the validation pile was kept out of training.

Stopping at the bottom of the validation curve is called **early stopping**. Let's compare the overtrained network with one stopped at the best epoch, on the training pile:

```
# Retrain from the same starting weights, but stop at the best epoch. With
# the same seed and the same data it follows exactly the same path, so this
```

```
# is the network as it was at that moment.
torch_manual_seed(1)
stopped = train_nn(to_x(train$temp), to_y(train$abundance),
                   hidden = 64, epochs = best_epoch)

# The training points, the truth, and both networks' curves
stopped_label = paste(best_epoch, "epochs")   # legend label, e.g. "95 epochs"
ggplot(train, aes(x = temp, y = abundance)) +
  geom_point(color = col_train, alpha = 0.6) +
  geom_line(data = grid, aes(y = true_curve(temp)), linewidth = 0.8) +
  geom_line(data = grid, aes(y = predict_nn(big$net, temp), color = "600 epochs"),
            linewidth = 1.1) +
  geom_line(data = grid, aes(y = predict_nn(stopped$net, temp), color = stopped_label),
            linewidth = 1.1) +
  scale_color_manual(values = setNames(c(col_valid, col_nn),
                                       c("600 epochs", stopped_label))) +
  labs(x = "Temperature (°C)", y = "Abundance index", color = NULL) +
  theme_minimal(base_size = 14) +
  theme(legend.position = "top")
```



The overtrained curve bends to chase individual points. The early-stopped one tracks the true niche.

Every choice is now made: architecture, epochs, everything. Only now do we open the drawer and score the **test** pile, once:

```
# The first and only time the test pile is used. Each model was trained on
# the training pile, and every choice was made with the validation pile.
tibble(
  model = c("Straight line", "Network, 600 epochs",
            paste("Network,", best_epoch, "epochs")),
  test_rmse = c(rmse(test$abundance, predict(lm(abundance ~ temp, data = train), test)),
                rmse(test$abundance, predict_nn(big$net, test$temp)),
                rmse(test$abundance, predict_nn(stopped$net, test$temp)))
)
```

```
# A tibble: 3 × 2
  model               test_rmse
  <chr>                   <dbl>
1 Straight line           15.0
2 Network, 600 epochs      9.11
3 Network, 95 epochs       7.03
```

4.1 Play with it: five experiments

This Shiny app trains the same network on the same kind of survey, with the amount of data and every hyperparameter in the sidebar. Press **Train** after changing settings, then compare the three scores and both loss curves. "Off from the true niche" is the score that matters most here, and it is only possible because we invented the species. The test pile never appears in the app, on purpose.

Take about ten minutes for these five experiments. Start each one from the app's opening settings (150 plots, 16 units, 300 epochs, learning rate 0.03, no weight decay, ReLU), and change only what it says.

1. **More complexity, same data.** Train with 2, then 16, then 64 hidden units. Does validation error keep improving as the network grows?
2. **Training too long.** Set 64 hidden units, then train for 100, 300 and 1,000 epochs. Watch the validation curve and the distance from the true niche.
3. **More data.** Keep 64 units and 1,000 epochs, and try 30, 150, 600 and 1,500 survey plots.
4. **Regularization.** Back to 150 plots, with 64 units and 1,000 epochs. Raise the weight decay from 0 to 0.01, then to 0.1.
5. **Is more complicated always better?** Look back at your best validation score from experiments 1, 2 and 4. Which settings won, and was it the biggest, longest-trained network?

Run the chunks above first, since the app uses `small`, `train_nn()` and `true_curve()`. If you have changed `train_nn()`, rerun everything above with Ctrl+Alt+P (Cmd+Option+P on a Mac) before starting the app.

```
library(shiny)
library(bslib)

# ---- A bigger survey to draw from --------------------------------------------
# The first 150 rows are this section's survey (`small`), so 150 plots in the
# app is exactly the data above. The rest are 1,350 extra plots of the same
# species, dealt into the same three piles in the same 60/20/20 proportions.
```

```
# The slider takes the first n rows, so more data only ever ADDS plots.
set.seed(10)
extra = tibble(temp = runif(1350, 0, 30)) |>
  mutate(abundance = true_curve(temp) + rnorm(n(), 0, 8),
         pile = sample(c("train", "validation", "test"), n(), replace = TRUE,
                       prob = c(0.6, 0.2, 0.2)))
pool = bind_rows(small, extra)   # 1,500 plots in all

# ---- UI: data and hyperparameters, three scores, two plots --------------------
ui = page_sidebar(
  title = "Neural network playground",
  sidebar = sidebar(
    # How much data: the most important setting of all
    sliderInput("n_samples", "Survey plots", min = 30, max = 1500,
                value = 150, step = 30),
    textOutput("pile_sizes"),   # how many plots landed in each pile
    # Every hyperparameter from the table in section 3, small enough for any laptop
    selectInput("hidden", "Hidden units",
                choices = c(1, 2, 4, 8, 16, 32, 64), selected = 16),
    sliderInput("epochs", "Epochs", min = 50, max = 1000, value = 300, step = 50),
   selectInput("lr", "Learning rate", choices = c(0.003, 0.03, 0.3), selected = 0.03),
    selectInput("decay", "Weight decay",
                choices = c(0, 0.0001, 0.001, 0.01, 0.1), selected = 0),
    selectInput("activation", "Activation",
                choices = c("ReLU (kinked)" = "relu", "tanh (smooth S)" = "tanh",
                            "Softplus (rounded ReLU)" = "softplus")),
    plotOutput("bend_plot", height = 140),   # a picture of the chosen bend
    # Nothing trains until this button is pressed
    actionButton("go", "Train", class = "btn-primary"),
    p("Bigger settings take a few seconds on a slow laptop.")
  ),
  # Top row: three scores, in abundance units
  layout_columns(
    fill = FALSE,
    value_box(title = "Training RMSE", value = textOutput("train_rmse")),
    value_box(title = "Validation RMSE", value = textOutput("valid_rmse")),
    value_box(title = "Off from the true niche", value = textOutput("truth_rmse"))
  ),
  # Bottom row: the fitted curve, and both loss curves
  layout_columns(
    card(plotOutput("fit_plot")),
    card(plotOutput("loss_plot"))
  )
)

# ---- Server -------------------------------------------------------------------
```

```
server = function(input, output, session) {

  # Everything that happens when Train is pressed:
  #   1. take the first n plots from the pool and split them into piles
  #   2. standardize with THIS training pile's mean and sd (no leakage)
  #   3. train a network with the chosen settings
  # Selects send their values as text, so as.integer() and as.numeric() turn
  # them back into numbers. bindEvent(): only retrain when the button is
  # pressed, not on every slider wiggle. ignoreNULL = FALSE also trains once
  # at startup, so the plots are not empty.
  model = reactive({
    survey = pool[seq_len(input$n_samples), ]
    tr = filter(survey, pile == "train")
    va = filter(survey, pile == "validation")

    # Standardizing numbers from this training pile only
    x_mu = mean(tr$temp);      x_s = sd(tr$temp)
    y_mu = mean(tr$abundance); y_s = sd(tr$abundance)
    sx = function(temp) torch_tensor(matrix((temp - x_mu) / x_s))
    sy = function(ab) torch_tensor(matrix((ab - y_mu) / y_s))

    torch_manual_seed(1)
    fit = train_nn(sx(tr$temp), sy(tr$abundance),
                   hidden = as.integer(input$hidden), epochs = input$epochs,
                   lr = as.numeric(input$lr), decay = as.numeric(input$decay),
                   activation = input$activation,
                   x_val = sx(va$temp), y_val = sy(va$abundance))

    # Hand back the piles, the loss history, and a prediction function
    list(train = tr, valid = va, history = fit$history,
         predict = function(temp) {
           as.numeric(with_no_grad(fit$net(sx(temp)))) * y_s + y_mu
         })
  }) |>
    bindEvent(input$go, ignoreNULL = FALSE)

  # The chosen bend, drawn as soon as the dropdown changes. No training
  # needed: it is just the activation function applied to -4 ... 4.
  output$bend_plot = renderPlot({
    z = seq(-4, 4, length.out = 201)
    out = switch(input$activation,
      relu     = nnf_relu(torch_tensor(z)),
      tanh     = torch_tanh(torch_tensor(z)),
      softplus = nnf_softplus(torch_tensor(z))
    )
    ggplot(tibble(z = z, out = as.numeric(out)), aes(z, out)) +
```

```
    geom_hline(yintercept = 0, color = "grey75") +
    geom_vline(xintercept = 0, color = "grey75") +
    geom_line(color = col_nn, linewidth = 1.2) +
    labs(x = "in", y = "out") +
    theme_minimal(base_size = 10)
})

# How the current survey was split
output$pile_sizes = renderText({
  m = model()
  sprintf("%d training and %d validation plots", nrow(m$train), nrow(m$valid))
})

# The three scores. All re-run whenever model() changes.
output$train_rmse = renderText({
  m = model()
  sprintf("%.2f", rmse(m$train$abundance, m$predict(m$train$temp)))
})
output$valid_rmse = renderText({
  m = model()
  sprintf("%.2f", rmse(m$valid$abundance, m$predict(m$valid$temp)))
})
# Only possible because we invented the species: how far the network's
# curve is from the real niche, across the whole temperature range
output$truth_rmse = renderText({
  m = model()
  sprintf("%.2f", rmse(true_curve(grid$temp), m$predict(grid$temp)))
})

# Left plot: both piles of points, the true niche, and the network's curve
output$fit_plot = renderPlot({
  m = model()
  dot_alpha = if (nrow(m$train) > 300) 0.35 else 0.7   # fade dots when crowded
  ggplot() +
    geom_point(data = m$train, aes(x = temp, y = abundance, color = "train"),
               alpha = dot_alpha) +
    geom_point(data = m$valid, aes(x = temp, y = abundance, color = "validation"),
               alpha = dot_alpha) +
    geom_line(data = grid, aes(x = temp, y = true_curve(temp)), linewidth = 0.8) +
    geom_line(data = tibble(temp = grid$temp, fit = m$predict(grid$temp)),
              aes(x = temp, y = fit), color = col_nn, linewidth = 1.2) +
    scale_color_manual(values = c(train = col_train, validation = col_valid)) +
    labs(x = "Temperature (°C)", y = "Abundance index", color = NULL) +
    theme_minimal(base_size = 14) + theme(legend.position = "top")
})
```

```
  # Right plot: training and validation loss after every epoch
  output$loss_plot = renderPlot({
    model()$history |>
      tidyr::pivot_longer(c(train, validation),
                          names_to = "pile", values_to = "loss") |>
      ggplot(aes(x = epoch, y = loss, color = pile)) +
      geom_line(linewidth = 1) +
      scale_color_manual(values = c(train = col_train, validation = col_valid)) +
      labs(x = "Epoch", y = "Loss", color = NULL) +
      theme_minimal(base_size = 14) + theme(legend.position = "top")
  })
}

shinyApp(ui, server)
```

Try it

In the playground, find settings where the validation curve turns back up. If you did not finish the five experiments above, finish them now. In experiment 3, why do both errors stop improving at around the same number?

Then switch the activation to tanh, with 64 hidden units and 1,000 epochs, and no weight decay. Is a smooth network safe from overfitting?

Back in the document, change `set.seed(2)` at the top of this section to `set.seed(3)`, then rerun the section. Does the best epoch stay the same? What does that tell you about trusting one validation pile of 30 points?

Ask the model

"I tuned my neural network by trying different settings and keeping whichever gave the lowest error on my test set. Then I reported that test error in my paper. What is wrong with this, and how should I split my data instead?"

4.2 Before you go: can you trust it?

Five things to take with you:

1. **A neural network is a flexible function fit by walking downhill.** With no hidden layer, it is linear regression.
2. **Hidden units and their bends find nonlinear shapes**, like a species niche, without being told the shape.
3. **The same flexibility lets it memorize noise.** Only data it never trained on can tell the two apart.
4. **Each pile has one job.** Fit the weights with training data, choose settings with validation data, and score once with test data. Compute any preprocessing, such as standardizing, from the training pile only.

5. **The split has to match the use.** Random piles were fine for our simulated plots. Real environmental data usually needs spatial, temporal or site-level separation.

The checklist at the end turns these into six questions to ask of any model you did not train yourself, including one a chatbot just wrote for you.

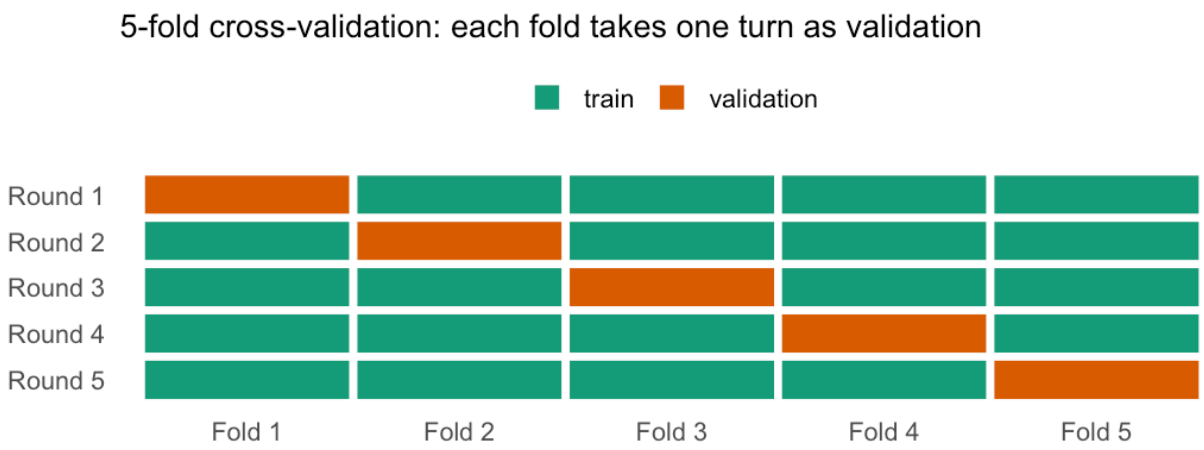
Sections 5 and 6 are optional extensions for after the workshop: k-fold cross-validation on real ozone data, and how recurrent and convolutional networks handle time and space.

5 k-fold cross-validation: a fair fight on real data (optional)

An optional extension for after the workshop. It trains 25 small networks, which takes about 3 seconds on a recent laptop and up to 15 on an older one.

The last Try it showed the weakness of one validation pile: with 30 points, the best setting depends on which 30 you happened to draw. Real ecological data is often that small. **k-fold cross-validation** fixes it by taking turns.

Split the non-test data into k chunks, called folds. Train on all but one fold and validate on the one left out. Repeat until every fold has been the validation pile once, then average. Every row gets used for validation exactly once, and you get k scores instead of one, so you can also see how much they vary.



Back to ozone, now with all three weather variables. The question: can a neural network beat a linear model at predicting ozone, fairly?

```
# The 111 complete days again, keeping ozone and the three weather inputs:
# sunlight (Solar.R, langleys), wind (mph) and temperature (°F)
aq = na.omit(airquality) |> select(Ozone, Solar.R, Wind, Temp)
preds = c("Solar.R", "Wind", "Temp")   # the input columns, by name

# ---- Step 1: lock away a test set before doing anything else -----------------
set.seed(4)
test_rows = sample(nrow(aq), 22)   # 22 random days, about 20%
aq_test = aq[test_rows, ]          # into the drawer
aq_dev  = aq[-test_rows, ]         # "development" data: everything else
```

```
c(development = nrow(aq_dev), test = nrow(aq_test))
```

```
development        test
         89          22
```

We need a network-fitting function for three inputs. Notice where the scaling numbers come from: the training data it is handed, and nothing else. Inside a cross-validation loop that is what keeps each fold's validation rows unseen.

```
# Fit a network that predicts ozone from the three weather inputs.
#   train_data: the rows to train on (a data frame)
#   hidden, decay, epochs, lr: hyperparameters, passed on to train_nn()
# Returns a FUNCTION that predicts ozone for any new rows. The function
# carries the training data's standardizing numbers inside it.
fit_ozone_nn = function(train_data, hidden = 16, decay = 0, epochs = 150, lr = 0.03) {

  # Standardizing numbers from THIS training data only
  mu = colMeans(train_data[preds]); s = sapply(train_data[preds], sd)   # one per input
  y_mu = mean(train_data$Ozone);     y_s = sd(train_data$Ozone)        # for the output

  # scale() standardizes every column at once, using the numbers we give it
  x = torch_tensor(scale(as.matrix(train_data[preds]), mu, s))   # rows x 3 inputs
  y = torch_tensor(matrix((train_data$Ozone - y_mu) / y_s))      # rows x 1 answer
  fit = train_nn(x, y, hidden = hidden, epochs = epochs, lr = lr, decay = decay)

  # Hand back a prediction function. It standardizes new rows with the
  # TRAINING numbers, predicts, and converts back to ppb of ozone.
  function(new_data) {
    x_new = torch_tensor(scale(as.matrix(new_data[preds]), mu, s))
    as.numeric(with_no_grad(fit$net(x_new))) * y_s + y_mu
  }
}
```

Step 2: five-fold cross-validation on the development data. The linear model gets the same folds as four versions of the network, so the comparison is fair.

```
# ---- Assign every development row to one of 5 folds, at random --------------
k = 5
set.seed(5)
# A fold number, 1 to 5, for each row, shuffled
folds = sample(rep(1:k, length.out = nrow(aq_dev)))
table(folds)                                          # about 18 days per fold
```

```
folds
 1  2  3  4  5
18 18 18 18 17
```

```
# ---- The network settings to compare: 2 sizes x 2 weight decays = 4 ----------
configs = expand.grid(hidden = c(8, 32), decay = c(0, 0.01))

# ---- The cross-validation loop -----------------------------------------------
results = list()                          # one row per model per fold goes in here
for (fold in 1:k) {
  # This round's piles: one fold validates, the other four train
  fold_train = aq_dev[folds != fold, ]
  fold_valid = aq_dev[folds == fold, ]

  # The linear model: train on the four folds, score on the fifth
  linear = lm(Ozone ~ Solar.R + Wind + Temp, data = fold_train)
  results[[length(results) + 1]] = tibble(
    fold = fold, model = "Linear model",
    rmse = rmse(fold_valid$Ozone, predict(linear, fold_valid))
  )

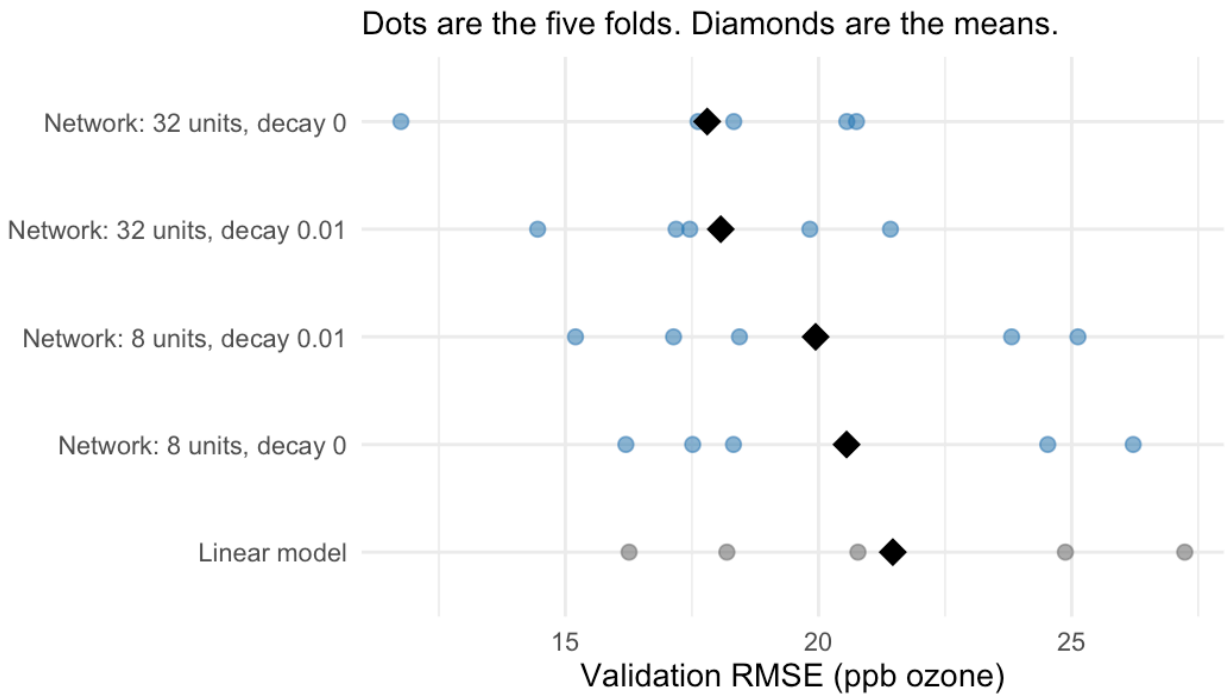
  # Each network setting: same training rows, same validation rows
  for (i in seq_len(nrow(configs))) {
    torch_manual_seed(fold)   # reproducible starting weights
    predict_ozone = fit_ozone_nn(fold_train, hidden = configs$hidden[i],
                                 decay = configs$decay[i])
    results[[length(results) + 1]] = tibble(
      fold = fold,
      model = sprintf("Network: %d units, decay %g",
                      configs$hidden[i], configs$decay[i]),
      rmse = rmse(fold_valid$Ozone, predict_ozone(fold_valid))
    )
  }
}
cv = bind_rows(results)   # 5 folds x 5 models = 25 rows

# ---- Summarize: average score and fold-to-fold spread for each model ---------
cv_summary = cv |>
  summarise(mean_rmse = mean(rmse), sd_rmse = sd(rmse), .by = model) |>
  arrange(mean_rmse)    # best (lowest error) first
cv_summary
```

```
# A tibble: 5 × 3
  model                          mean_rmse sd_rmse
  <chr>                              <dbl>   <dbl>
```

```
1 Network: 32 units, decay 0         17.8    3.65
2 Network: 32 units, decay 0.01      18.1    2.67
3 Network: 8 units, decay 0.01       19.9    4.32
4 Network: 8 units, decay 0          20.6    4.50
5 Linear model                       21.5    4.56
```

```
# One row per model, ordered by average error. Small dots are the five folds,
# and the black diamond is their mean.
ggplot(cv, aes(x = rmse, y = reorder(model, -rmse, FUN = mean))) +
  geom_point(aes(color = model == "Linear model"), size = 2.5, alpha = 0.6) +
  geom_point(data = cv_summary, aes(x = mean_rmse, y = model), shape = 18, size = 5) +
  scale_color_manual(values = c(`TRUE` = col_linear, `FALSE` = col_nn),
                     guide = "none") +
  labs(x = "Validation RMSE (ppb ozone)", y = NULL,
       subtitle = "Dots are the five folds. Diamonds are the means.") +
  theme_minimal(base_size = 13)
```



Each row of dots is five honest scores. The spread matters as much as the mean. If two models' dots overlap heavily, the data cannot tell them apart.

Step 3: choose the best setting **using cross-validation only**, refit it on all the development data, and score the test set once.

```
# ---- Choose: the network setting with the lowest cross-validation error ------
best = cv_summary |> filter(model != "Linear model") |> slice(1)
```

```
# Find that setting's row in configs, so we can reuse its numbers
config_names = sprintf("Network: %d units, decay %g", configs$hidden, configs$decay)
best_config  = configs[config_names == best$model, ]
best$model
```

```
[1] "Network: 32 units, decay 0"
```

```
# ---- Refit both finalists on ALL the development data ------------------------
final_linear = lm(Ozone ~ Solar.R + Wind + Temp, data = aq_dev)
torch_manual_seed(1)
final_nn = fit_ozone_nn(aq_dev, hidden = best_config$hidden, decay = best_config$decay)

# ---- Open the drawer: score the test set, once -------------------------------
tibble(
  model = c("Linear model", best$model),
  test_rmse = c(rmse(aq_test$Ozone, predict(final_linear, aq_test)),
                rmse(aq_test$Ozone, final_nn(aq_test)))
)
```

```
# A tibble: 2 × 2
  model                      test_rmse
  <chr>                          <dbl>
1 Linear model                    20.9
2 Network: 32 units, decay 0      16.4
```

A few honest caveats. The test set is only 22 days, so its score is noisy. The cross-validation means are the stronger evidence, and the test is the final sanity check. And the linear model here is the plain textbook one. A linear model with the right transformations can close much of the gap. The network's advantage is that it finds those shapes without being told, and its costs are the tuning and the interpretability.

> ! A trap ecologists fall into
>
> Random folds assume every row is independent. Field data rarely is. Days next to each other have similar weather, plots next to each other share soil, and repeat visits share a site. When near-duplicate rows land in both training and validation, the model is graded on questions it has already seen, and the score looks better than it will be in a new place or a new year. The fix is to split by **blocks**: by year, by site, or by spatial block. *Roberts et al. (2017), Cross-validation strategies for data with temporal, spatial, hierarchical, or phylogenetic structure,* in Ecography, is the standard reference.

> **Try it**
>
> Give the linear model a fairer shot. In the cross-validation loop, change its formula to `Ozone ~ Solar.R + Wind + Temp + I(Temp^2) + I(Wind^2)` and rerun. Does the gap to the network shrink? Which result would you report?

> **Ask the model**
>
> Paste the cross-validation chunk and ask: "Check this for data leakage. Is anything computed from the full dataset, or from the validation fold, before the model is trained on each fold?" Then ask it what would change if these rows were daily measurements from the same summer.

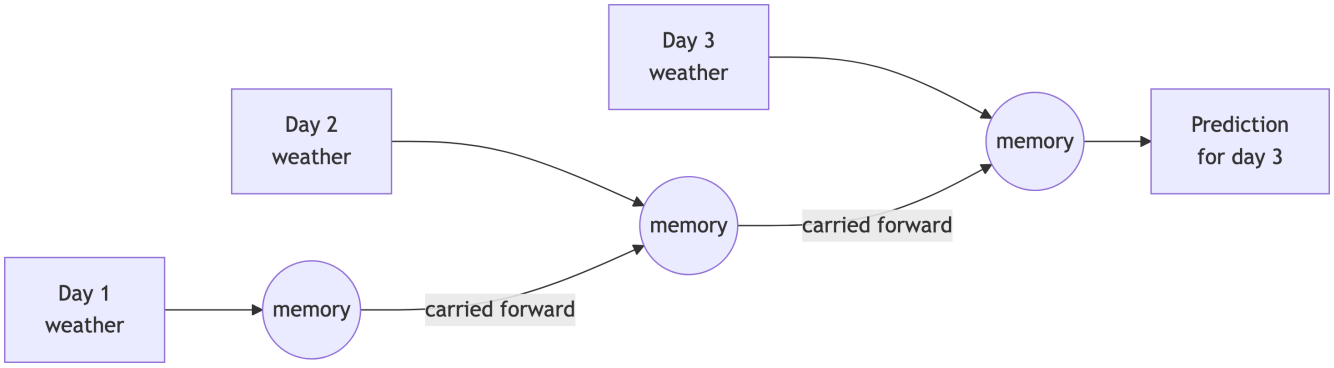
6 Beyond one hidden layer: time and space (optional)

An optional extension for after the workshop: two conceptual tours, with small demonstrations rather than trained models.

Everything today used one design: inputs in, one hidden layer, one output. Two other designs dominate environmental work. Both are built from the same parts you now know: weights, bends, the five-line training loop and the three piles. What changes is how the network is wired to match the shape of the data. The architecture is partly chosen by the science: whether your observations are linked through time, through space, or not at all.

6.1 Recurrent networks: memory for time series

Many ecological responses depend on history, not just today. Streamflow depends on weeks of rain and on snow that fell months ago. Soil moisture depends on the last month. Leaf-out depends on warmth accumulated since winter. A **recurrent neural network** (RNN) reads a time series one step at a time and carries a **hidden state** forward: a memory it updates every step. The same small network, with the same weights, runs at every step.



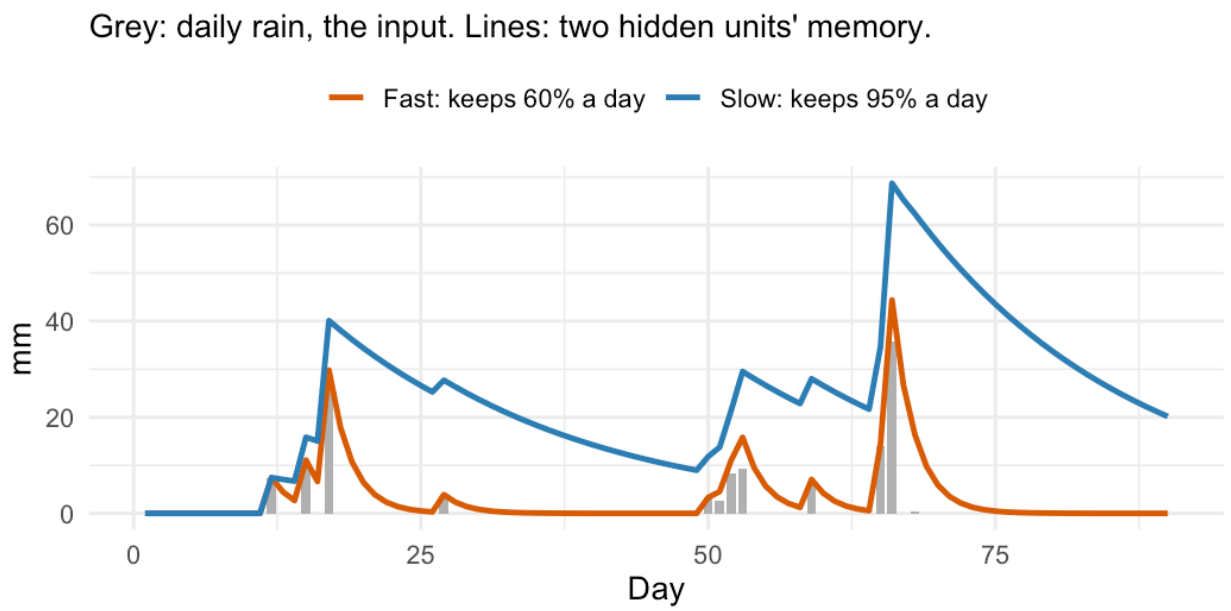
Here is the idea with no training at all. Each hidden unit below is a leaky bucket. Every day it keeps a fraction of yesterday's memory and adds today's rain. A slow unit remembers rain for weeks, and a fast unit forgets it in days.

```r
# ---- Ninety days of rain ------------------------------------------------------
set.seed(8)
day  = 1:90
rain = ifelse(runif(90) < 0.15, rexp(90, rate = 1 / 12), 0)   # mostly dry, some storms

# ---- One recurrent unit, written out by hand ----------------------------------
# memory today = keep x memory yesterday + rain today
# `keep` is a weight. A real RNN learns it, along with a bend and many more units.
run_unit = function(rain, keep) {
  memory = numeric(length(rain))
  for (t in seq_along(rain)) {
    yesterday = if (t == 1) 0 else memory[t - 1]   # nothing remembered on day 1
    memory[t] = keep * yesterday + rain[t]
  }
  memory
}

# ---- Two units with different memories ----------------------------------------
units = bind_rows(
  tibble(day = day, memory = run_unit(rain, keep = 0.95),
         unit = "Slow: keeps 95% a day"),
  tibble(day = day, memory = run_unit(rain, keep = 0.60),
         unit = "Fast: keeps 60% a day")
)

ggplot() +
  geom_col(data = tibble(day = day, rain = rain), aes(day, rain),
           fill = "grey70", width = 0.8) +                       # the input
  geom_line(data = units, aes(day, memory, color = unit), linewidth = 1.1) +
  scale_color_manual(values = c(col_valid, col_nn)) +
  labs(x = "Day", y = "mm", color = NULL,
       subtitle = "Grey: daily rain, the input. Lines: two hidden units' memory.") +
  theme_minimal(base_size = 13) +
  theme(legend.position = "top")
```



A trained RNN has dozens of units like these. It learns how much each one keeps and how much each input adds, and it passes the result through the same bends you saw in section 3. Slow units can track something like a snowpack and fast units something like a storm. **LSTMs** and **GRUs** are RNNs with gates: learned switches that decide what to keep and what to forget. torch has them as `nn_lstm()` and `nn_gru()`.

The three piles change shape for time series. Train on the earlier years, validate on later years, and test on the most recent ones. Never deal days into random piles: the model would see tomorrow while learning to predict today.

6.2 Convolutional networks: neighborhoods for maps and images

Gridded data, such as satellite images, climate rasters and camera-trap photos, have spatial structure. A pixel means more alongside its neighbors. A **convolutional neural network** (CNN) slides a small window, called a **filter**, across the grid. The filter's weights are applied to every neighborhood in turn, so a pattern is recognized wherever it occurs.

Here is one filter on a made-up landscape: a patch of forest in open ground. The filter's nine weights are set by hand to detect edges. This is torch's real convolution function, the same one a CNN uses.

```
# ---- A 16 x 16 landscape: tree cover, high in a forest patch, low outside ---
set.seed(7)
n = 16
patch = outer(1:n, 1:n, function(r, c) sqrt((c - 8.5)^2 + ((r - 8.5) / 1.2)^2) < 5)
land  = ifelse(patch, 0.85, 0.15) + matrix(rnorm(n * n, 0, 0.04), n, n)

# ---- One 3 x 3 filter: 9 weights. These ones detect edges --------------------
# The centre cell minus its eight neighbours: about zero where everything
# around is the same, large where cover changes, as at a forest edge.
filt = matrix(c(-1, -1, -1,
                -1,  8, -1,
```

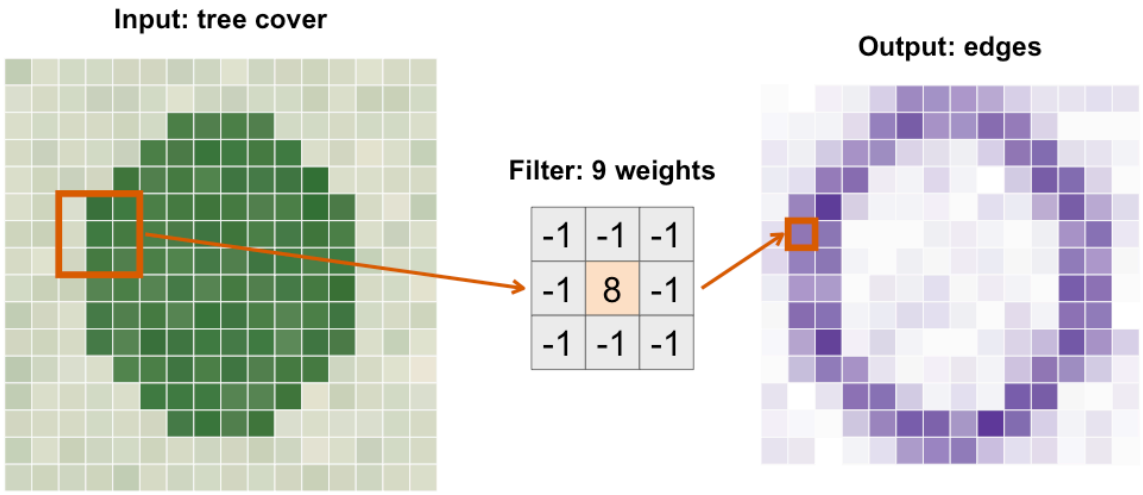
```
               -1, -1, -1), 3, 3, byrow = TRUE)

# ---- Slide the filter over every 3 x 3 neighbourhood -------------------------
# torch wants 4 dimensions: (images, channels, rows, columns). One of each here.
edges = nnf_conv2d(torch_tensor(land)$view(c(1, 1, n, n)),
                   torch_tensor(filt)$view(c(1, 1, 3, 3)))
edges = abs(as_array(edges$view(c(n - 2, n - 2))))   # 14 x 14: edges lose a cell

# ---- Draw input, filter and output side by side ------------------------------
# Turn a matrix into tiles placed at (x0, y0), each `size` units across,
# with colors from a two-color ramp
tiles = function(m, x0, y0, low, high, lims = range(m), size = 1) {
  cells = expand.grid(r = seq_len(nrow(m)), c = seq_len(ncol(m)))
  value = m[cbind(cells$r, cells$c)]
  shade = round(99 * (value - lims[1]) / diff(lims)) + 1
  tibble(x = x0 + size * cells$c, y = y0 - size * cells$r, value = value,
         fill = colorRampPalette(c(low, high))(100)[pmin(pmax(shade, 1), 100)])
}
input_tiles  = tiles(land,  0,      0,  "#f3efe0", "#2d6a2e", c(0, 1))
filter_tiles = tiles(filt,  n + 3,  -5, "grey92",  "#fde0c5", size = 2)  # drawn 2x
output_tiles = tiles(edges, n + 12, -1, "white",   "#5e3c99")
r0 = 6; c0 = 2   # the highlighted window: input rows 6 to 8, columns 2 to 4
pointer = arrow(length = unit(0.18, "cm"))

ggplot() +
  geom_tile(data = input_tiles,  aes(x, y, fill = fill),
            color = "white", linewidth = 0.2) +
  geom_tile(data = filter_tiles, aes(x, y, fill = fill),
            width = 2, height = 2, color = "grey40") +
  geom_text(data = filter_tiles, aes(x, y, label = value), size = 5) +
  geom_tile(data = output_tiles, aes(x, y, fill = fill),
            color = "white", linewidth = 0.2) +
  scale_fill_identity() +
  # the window in the input, and the one output cell it produces
  annotate("rect", xmin = c0 + 0.5, xmax = c0 + 3.5,
           ymin = -r0 - 2.5, ymax = -r0 + 0.5,
           fill = NA, color = col_valid, linewidth = 1.3) +
  annotate("rect", xmin = n + 11.5 + c0, xmax = n + 12.5 + c0,
           ymin = -1.5 - r0, ymax = -0.5 - r0,
           fill = NA, color = col_valid, linewidth = 1.3) +
  # arrows: window -> filter -> output cell
  annotate("segment", x = c0 + 3.7, xend = n + 3.7, y = -r0 - 1, yend = -9,
           color = col_valid, linewidth = 0.7, arrow = pointer) +
  annotate("segment", x = n + 10.3, xend = n + 11.3 + c0, y = -9, yend = -1 - r0,
           color = col_valid, linewidth = 0.7, arrow = pointer) +
  # titles
```

```
  annotate("text", x = n / 2 + 0.5, y = 1, label = "Input: tree cover",
           fontface = "bold") +
  annotate("text", x = n + 7, y = -4.6, label = "Filter: 9 weights",
           fontface = "bold") +
  annotate("text", x = n + 19.5, y = 0, label = "Output: edges",
           fontface = "bold") +
  coord_equal() +
  theme_void()
```



The orange window covers nine cells. Each is multiplied by the matching filter weight, and the nine products are added up into one output cell. Slide the window one cell and repeat, and the output map lights up wherever forest meets open ground.

A CNN learns its filter weights with the same training loop as everything else today, instead of having them set by hand. Real CNNs learn dozens of filters in each layer and stack the layers, so later filters find patterns of patterns: edges, then patches, then whole landscapes. A few ordinary layers like ours at the end turn all of that into a prediction.



For maps, the three piles are spatial blocks, not random pixels. Neighboring pixels nearly duplicate each other, so a model validated on pixels next to its training pixels is graded on answers it has already seen.

Ask the model

"I want to predict daily streamflow at one river gauge from 20 years of daily weather. Should I use a recurrent network, a convolutional network, or a plain network with one hidden layer? How should I split the 20 years into training, validation and test data, and why can't I shuffle the days at random?"

6.3 Tools for real projects

Everything here was written by hand so you could see it. For real projects you would reach for tools that do the bookkeeping:

- **tidymodels** (`rsample`, `tune`, `yardstick`) handles splits, cross-validation folds, tuning grids and metrics, including spatial folds through the `spatialsample` package.
- **brulee** fits exactly this kind of network inside tidymodels, using torch underneath.
- **luz** wraps the torch training loop with early stopping, learning-rate schedules and progress bars. It trains recurrent and convolutional networks built from `nn_gru()`, `nn_lstm()` and `nn_conv2d()` the same way.

Appendix: reading a model you did not train

Use this on your own models, on models in papers, and above all on code a chatbot writes for you.

1. **Does it beat a simple baseline?** A network that cannot beat a linear model or a seasonal average is not earning its complexity.
2. **What was each pile used for?** Training fits weights, validation chooses settings, test scores once. Look for any place the test set influenced a choice.
3. **Where could information leak?** Scaling, filling missing values, or choosing variables using all the data. Near-duplicate rows across piles.
4. **Do the splits match the use?** If the model will predict new sites or new years, it should be validated on held-out sites or years.
5. **How variable are the scores?** One validation number hides a lot. Fold-to-fold spread shows whether a difference is real.
6. **Were the hyperparameters tuned, and how?** Default settings can be fine. Settings tuned on the test set are not.

If you can answer those for a model, you can direct a chatbot to fix it. If you cannot, you are trusting its defaults.

Resources

- torch for R: documentation, tutorials and articles.
- Tidy Modeling with R, Kuhn and Silge. Free. The chapters on resampling and tuning cover today's ideas in production form.
- An Introduction to Statistical Learning, James, Witten, Hastie and Tibshirani. Free. Chapter 10 covers neural networks, and chapter 5 covers cross-validation.
- Dive into Deep Learning. Free. Clear chapters on convolutional and recurrent networks, with diagrams.
- 3Blue1Brown: Neural networks. The best visual explanation of gradient descent and backpropagation.
- Roberts et al. (2017), *Cross-validation strategies for data with temporal, spatial, hierarchical, or phylogenetic structure*, Ecography 40: 913–929.