grokking
Deep Learning

DISCLAIMER:

ReadMe website is intended for academic and demonstration purposes only. We’re only showing a preview of the book to respect the author’s copyright. Thank you for your understanding!

- Group 4: The Classified

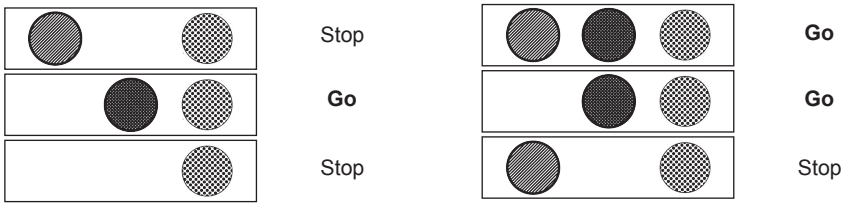
about the author

Andrew Trask is the founding member of Digital Reasoning's machine learning lab, where deep learning approaches to natural language processing, image recognition, and audio transcription are being researched. Within several months, Andrew and his research partner exceeded best published results in sentiment classification and part-of-speech tagging. He trained the world's largest artificial neural network with over 160 billion parameters, the results of which he presented with his coauthor at The International Conference on Machine Learning. Those results were published in the *Journal of Machine Learning*. He is currently the product manager of text and audio analytics at Digital Reasoning, responsible for driving the analytics roadmap for the Synthesys cognitive computing platform, for which deep learning is a core competency.

Nonparametric learning

Oversimplified: Counting-based methods

Nonparametric learning is a class of algorithm wherein the number of parameters is based on data (instead of predefined). This lends itself to methods that generally count in one way or another, thus increasing the number of parameters based on the number of items being counted within the data. In the supervised setting, for example, a nonparametric model might count the number of times a particular color of streetlight causes cars to "go." After counting only a few examples, this model would then be able to predict that *middle* lights always (100%) cause cars to go, and *right* lights only sometimes (50%) cause cars to go.



Notice that this model would have three parameters: three counts indicating the number of times each colored light turned on and cars would go (perhaps divided by the number of total observations). If there were five lights, there would be five counts (five parameters). What makes this simple model *nonparametric* is this trait wherein the number of parameters changes based on the data (in this case, the number of lights). This is in contrast to parametric models, which start with a set number of parameters and, more important, can have more or fewer parameters purely at the discretion of the scientist training the model (regardless of data).

A close eye might question this idea. The parametric model from before seemed to have a knob for each input datapoint. Most parametric models still have to have some sort of *input* based on the number of classes in the data. Thus you can see that there is a *gray area* between parametric and nonparametric algorithms. Even parametric algorithms are somewhat influenced by the number of classes in the data, even if they aren't explicitly counting patterns.

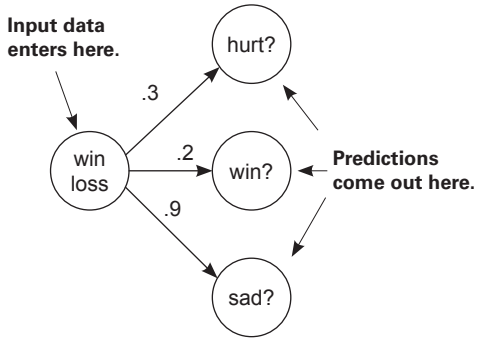
This also illuminates that *parameters* is a generic term, referring only to the set of numbers used to model a pattern (without any limitation on how those numbers are used). Counts are parameters. Weights are parameters. Normalized variants of counts or weights are parameters. Correlation coefficients can be parameters. The term refers to the set of numbers used to model a pattern. As it happens, deep learning is a class of parametric models. We won't discuss nonparametric models further in this book, but they're an interesting and powerful class of algorithm.

Making a prediction with multiple outputs

Neural networks can also make multiple predictions using only a single input.

Perhaps a simpler augmentation than multiple inputs is multiple outputs. Prediction occurs the same as if there were three disconnected single-weight neural networks.

❶ An empty network with multiple outputs

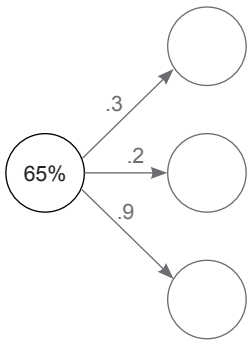


Instead of predicting just whether the team won or lost, you're also predicting whether the players are happy or sad and the percentage of team members who are hurt. You make this prediction using only the current win/loss record.

```
weights = [0.3, 0.2, 0.9]

def neural_network(input, weights):

    pred = ele_mul(input,weights)

    return pred
```

The most important comment in this setting is to notice that the three predictions are completely separate. Unlike neural networks with multiple inputs and a single output, where the prediction is undeniably connected, this network truly behaves as three independent components, each receiving the same input data. This makes the network simple to implement.

❷ Inserting one input datapoint




```
wlrec = [0.65, 0.8, 0.8, 0.9]

input = wlrec[0]

pred = neural_network(input,weights)
```

Hot and cold learning

This is perhaps the simplest form of learning.

Execute the following code in your Jupyter notebook. (New neural network modifications are in **bold**.) This code attempts to correctly predict 0.8:

```
weight = 0.5
input = 0.5
goal_prediction = 0.8

step_amount = 0.001

for iteration in range(1101):

    prediction = input * weight
    error = (prediction - goal_prediction) ** 2

    print("Error:" + str(error) + " Prediction:" + str(prediction))

    up_prediction = input * (weight + step_amount)
    up_error = (goal_prediction - up_prediction) ** 2

    down_prediction = input * (weight - step_amount)
    down_error = (goal_prediction - down_prediction) ** 2

    if(down_error < up_error):
        weight = weight - step_amount

    if(down_error > up_error):
        weight = weight + step_amount
```

How much to move the weights each iteration

Repeat learning many times so the error can keep getting smaller.

Try up!

Try down!

If down is better, go down!

If up is better, go up!

When I run this code, I see the following output:

```
Error:0.3025 Prediction:0.25
Error:0.30195025 Prediction:0.2505
            ....
Error:2.50000000033e-07 Prediction:0.7995
Error:1.07995057925e-27 Prediction:0.8
```

The last step correctly predicts 0.8!

Breaking gradient descent

Just give me the code!

```
weight = 0.5
goal_pred = 0.8
input = 0.5

for iteration in range(20):
    pred = input * weight
    error = (pred - goal_pred) ** 2
    delta = pred - goal_pred
    weight_delta = input * delta
    weight = weight - weight_delta
    print("Error:" + str(error) + " Prediction:" + str(pred))
```

When I run this code, I see the following output:

```
Error:0.3025 Prediction:0.25
Error:0.17015625 Prediction:0.3875
Error:0.095712890625 Prediction:0.490625
                    ...

Error:1.7092608064e-05 Prediction:0.79586567925
Error:9.61459203602e-06 Prediction:0.796899259437
Error:5.40820802026e-06 Prediction:0.797674444578
```

Now that it works, let's break it. Play around with the starting `weight`, `goal_pred`, and `input` numbers. You can set them all to just about anything, and the neural network will figure out how to predict the output given the input using the weight. See if you can find some combinations the neural network can't predict. I find that trying to break something is a great way to learn about it.

Let's try setting `input` equal to 2, but still try to get the algorithm to predict 0.8. What happens? Take a look at the output:

```
Error:0.04 Prediction:1.0
Error:0.36 Prediction:0.2
Error:3.24 Prediction:2.6
           ...

Error:6.67087267987e+14 Prediction:-25828031.8
Error:6.00378541188e+15 Prediction:77484098.6
Error:5.40340687069e+16 Prediction:-232452292.6
```

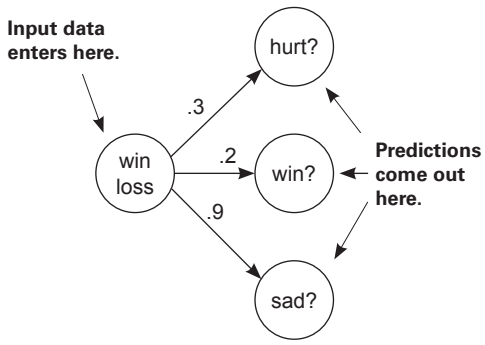
Whoa! That's not what you want. The predictions exploded! They alternate from negative to positive and negative to positive, getting farther away from the true answer at every step. In other words, every update to the weight overcorrects. In the next section, you'll learn more about how to combat this phenomenon.

Gradient descent learning with multiple outputs

Neural networks can also make multiple predictions using only a single input.

Perhaps this will seem a bit obvious. You calculate each `delta` the same way and then multiply them all by the same, single input. This becomes each weight's `weight_delta`. At this point, I hope it's clear that a simple mechanism (stochastic gradient descent) is consistently used to perform learning across a wide variety of architectures.

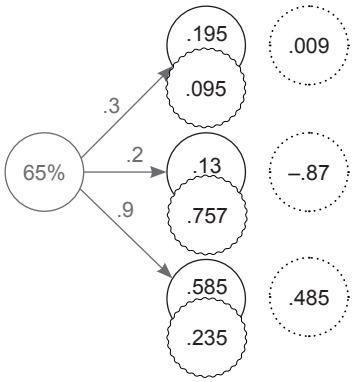
❶ An empty network with multiple outputs



Instead of predicting just whether the team won or lost, now you're also predicting whether they're happy or sad *and* the percentage of the team members who are hurt. You're making this prediction using only the current win/loss record.

```
weights = [0.3, 0.2, 0.9]

def neural_network(input, weights):

    pred = ele_mul(input,weights)

    return pred
```

❷ PREDICT: Making a prediction and calculating error and delta




```
wlrec = [0.65, 1.0, 1.0, 0.9]

hurt  = [0.1, 0.0, 0.0, 0.1]
win   = [  1,   1,   0,   1]
sad   = [0.1, 0.0, 0.1, 0.2]

input = wlrec[0]
true = [hurt[0], win[0], sad[0]]

pred = neural_network(input,weights)

error = [0, 0, 0]
delta = [0, 0, 0]

for i in range(len(true)):

    error[i] = (pred[i] - true[i]) ** 2
    delta[i] = pred[i] - true[i]
```

Learning the whole dataset

The neural network has been learning only one streetlight. Don't we want it to learn them all?

So far in this book, you've trained neural networks that learned how to model a single training example (`input -> goal_pred` pair). But now you're trying to build a neural network that tells you whether it's safe to cross the street. You need it to know more than one streetlight. How do you do this? You train it on all the streetlights at once:

```
import numpy as np

weights = np.array([0.5,0.48,-0.7])
alpha = 0.1

streetlights = np.array( [[ 1, 0, 1 ],
                          [ 0, 1, 1 ],
                          [ 0, 0, 1 ],
                          [ 1, 1, 1 ],
                          [ 0, 1, 1 ],
                          [ 1, 0, 1 ] ] )

walk_vs_stop = np.array( [ 0, 1, 0, 1, 1, 0 ] )

input = streetlights[0]
goal_prediction = walk_vs_stop[0]

for iteration in range(40):
    error_for_all_lights = 0
    for row_index in range(len(walk_vs_stop)):
        input = streetlights[row_index]
        goal_prediction = walk_vs_stop[row_index]

        prediction = input.dot(weights)

        error = (goal_prediction - prediction) ** 2
        error_for_all_lights += error

        delta = prediction - goal_prediction
        weights = weights - (alpha * (input * delta))
        print("Prediction:" + str(prediction))
    print("Error:" + str(error_for_all_lights) + "\n")
```

[1,0,1]

Equals 0 (stop)

```
Error:2.6561231104
Error:0.962870177672
...
Error:0.000614343567483
Error:0.000533736773285
```

Backpropagation in code

You can learn the amount that each weight contributes to the final error.

At the end of the previous chapter, I made an assertion that it would be important to memorize the two-layer neural network code so you could quickly and easily recall it when I reference more-advanced concepts. This is when that memorization matters.

The following listing is the new learning code, and it's essential that you recognize and understand the parts addressed in the previous chapters. If you get lost, go to chapter 5, memorize the code, and then come back. It will make a big difference someday.

```
import numpy as np

np.random.seed(1)

def relu(x):
    return (x > 0) * x

def relu2deriv(output):
    return output>0

alpha = 0.2
hidden_size = 4

weights_0_1 = 2*np.random.random((3,hidden_size)) - 1
weights_1_2 = 2*np.random.random((hidden_size,1)) - 1

for iteration in range(60):
   layer_2_error = 0
   for i in range(len(streetlights)):
      layer_0 = streetlights[i:i+1]
      layer_1 = relu(np.dot(layer_0,weights_0_1))
      layer_2 = np.dot(layer_1,weights_1_2)

      layer_2_error += np.sum((layer_2 - walk_vs_stop[i:i+1]) ** 2)

      layer_2_delta = (walk_vs_stop[i:i+1] - layer_2)
      layer_1_delta=layer_2_delta.dot(weights_1_2.T)*relu2deriv(layer_1)

      weights_1_2 += alpha * layer_1.T.dot(layer_2_delta)
      weights_0_1 += alpha * layer_0.T.dot(layer_1_delta)

   if(iteration % 10 == 9):
      print("Error:" + str(layer_2_error))
```

Returns x if x > 0; returns 0 otherwise

Returns 1 for input > 0; returns 0 otherwise

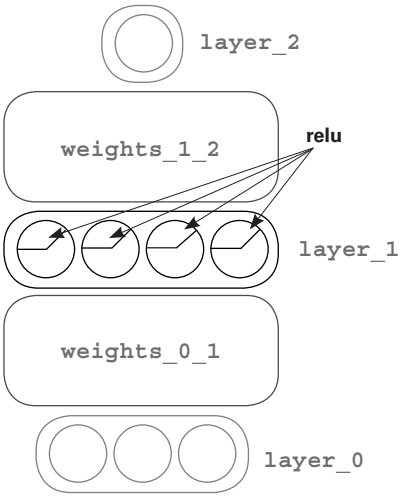
This line computes the delta at layer_1 given the delta at layer_2 by taking the layer_2_ delta and multiplying it by its connecting weights_1_2.

Believe it or not, the only truly new code is in bold. Everything else is fundamentally the same as in previous pages. The `relu2deriv` function returns 1 when `output` > 0; otherwise, it returns 0. This is the *slope* (the *derivative*) of the `relu` function. It serves an important purpose, as you'll see in a moment.

What is an activation function?

It's a function applied to the neurons in a layer during prediction.

An *activation function* is a function applied to the neurons in a layer during prediction. This should seem very familiar, because you've been using an activation function called `relu` (shown here in the three-layer neural network). The `relu` function had the effect of turning all negative numbers to 0.



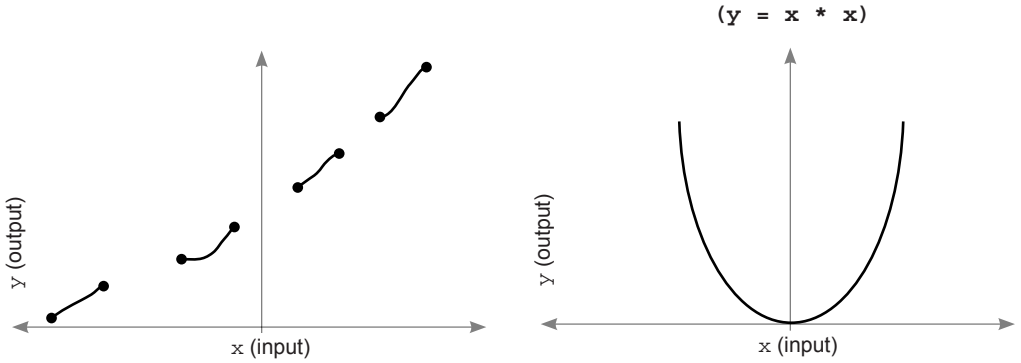
Oversimplified, an activation function is any function that can take one number and return another number. But there are an infinite number of functions in the universe, and not all them are useful as activation functions.

There are several constraints on what makes a function an activation function. Using functions outside of these constraints is usually a bad idea, as you'll see.

Constraint 1: The function must be continuous and infinite in domain.

The first constraint on what makes a proper activation function is that it must have an output number for *any* input. In other words, you shouldn't be able to put in a number that doesn't have an output for some reason.

A bit overkill, but see how the function on the left (four distinct lines) doesn't have `y` values for every `x` value? It's defined in only four spots. This would make for a horrible activation function. The function on the right, however, is continuous and infinite in domain. There is no input (`x`) for which you can't compute an output (`y`).

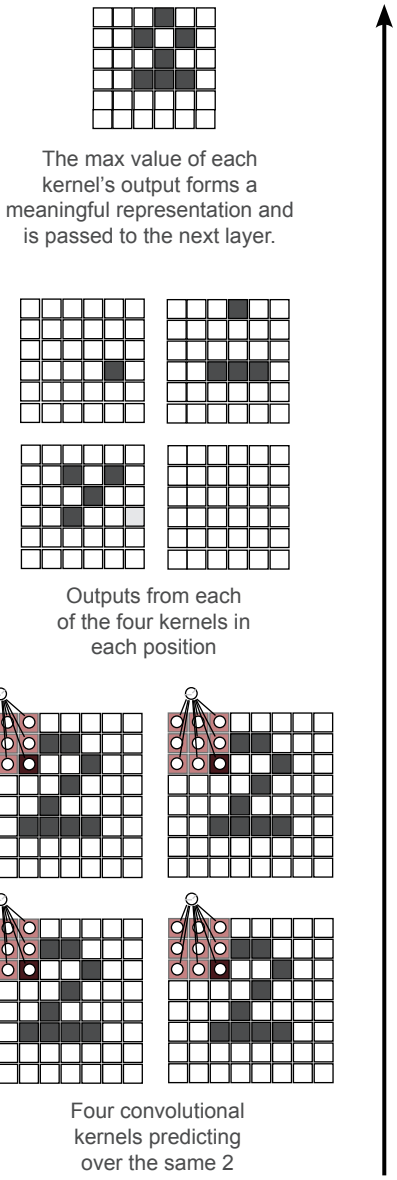


At bottom-right are four different convolutional kernels processing the same 8 × 8 image of a 2. Each kernel results in a 6 × 6 prediction matrix. The result of the convolutional layer with four 3 × 3 kernels is four 6 × 6 prediction matrices. You can either sum these matrices elementwise (sum pooling), take the mean elementwise (mean pooling), or compute the elementwise maximum value (max pooling).

The last version turns out to be the most popular: for each position, look into each of the four kernel's outputs, find the max, and copy it into a final 6 × 6 matrix as pictured at upper-right of this page. This final matrix (and only this matrix) is then forward propagated into the next layers.

There are a few things to notice in these figures. First, the bottom-right kernel forward propagates a 1 only if it's focused on a horizontal line segment. The bottom-left kernel forward propagates a 1 only if it's focused on a diagonal line pointing upward and to the right. Finally, the bottom-right kernel didn't identify any patterns that it was trained to predict.

It's important to realize that this technique allows each kernel to learn a particular pattern and then search for the existence of that pattern somewhere in the image. A single, small set of weights can train over a much larger set of training examples, because even though the dataset hasn't changed, each mini-kernel is forward propagated multiple times on multiple segments of data, thus changing the ratio of weights to datapoints on which those weights are being trained. This has a powerful impact on the network, drastically reducing its ability to overfit to training data and increasing its ability to generalize.



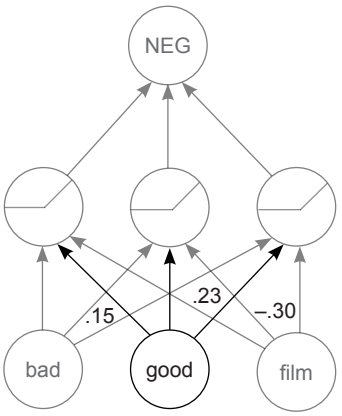
correlation summarization. If you can construct two examples with an identical hidden layer, one with the pattern you find interesting and one without, the network is unlikely to find that pattern.

As you just learned, a hidden layer fundamentally groups the previous layer's data. At a granular level, each neuron classifies a datapoint as either subscribing or not subscribing to its group. At a higher level, two datapoints (movie reviews) are similar if they subscribe to many of the same groups. Finally, two inputs (words) are similar if the weights linking them to various hidden neurons (a measure of each word's group affinity) are similar. Given this knowledge, in the previous neural network, what should you observe in the weights going into the hidden neurons from the words?

What should you see in the weights connecting words and hidden neurons?

Here's a hint: words that have a similar predictive power should subscribe to similar groups (hidden neuron configurations). What does this mean for the weights connecting each word to each hidden neuron?

Here's the answer. Words that correlate with similar labels (positive or negative) will have similar weights connecting them to various hidden neurons. This is because the neural network learns to bucket them into similar hidden neurons so that the final layer (`weights_1_2`) can make the correct positive or negative predictions.



The three bold weights for "good" form the embedding for "good." They reflect how much the term "good" is a member of each group (hidden neuron). Words with similar predictive power have similar word embeddings (weight values).

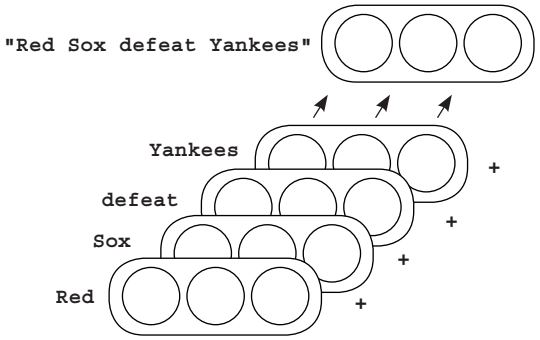
You can see this phenomenon by taking a particularly positive or negative word and searching for the other words with the most similar weight values. In other words, you can take each word and see which other words have the most similar weight values connecting them to each hidden neuron (to each group).

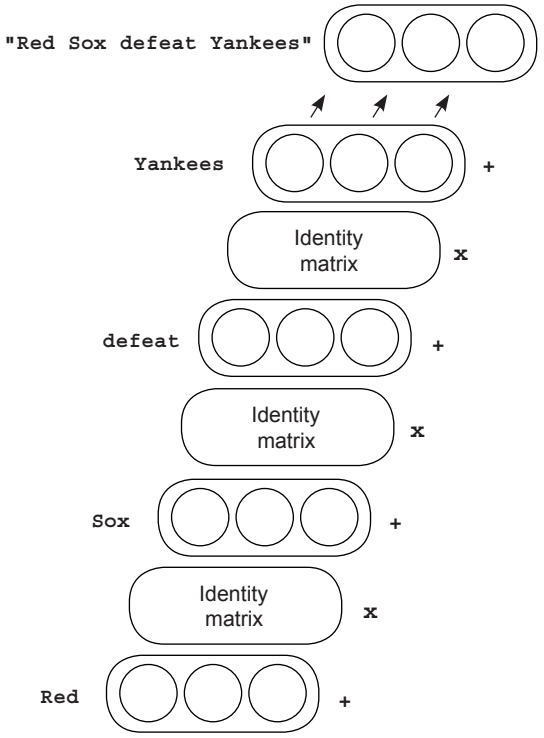
Words that subscribe to similar groups will have similar predictive power for positive or negative labels. As such, words that subscribe to similar groups, having similar weight values, will also have similar meaning. Abstractly, in terms of neural networks, a neuron has similar meaning to other neurons in the same layer if and only if it has similar weights connecting it to the next and/or previous layers.

Using identity vectors to sum word embeddings

Let's implement the same logic using a different approach.

You may think identity matrices are useless. What's the purpose of a matrix that takes a vector and outputs that same vector? In this case, we'll use it as a teaching tool to show how to set up a more complicated way of summing the word embeddings so the neural network can take order into account when generating the final sentence embedding. Let's explore another way of summing embeddings.





This is the standard technique for summing multiple word embeddings together to form a sentence embedding (dividing by the number of words gives the average sentence embedding). The example on the right adds a step *between* each sum: vector-matrix multiplication by an identity matrix.

The vector for "Red" is multiplied by an identity matrix, and then the output is summed with the vector for "Sox," which is then vector-matrix multiplied by the identity matrix and added to the vector for "defeat," and so on throughout the sentence. Note that because the vector-matrix multiplication by the identity matrix returns the same vector that goes into it, the process on the right yields *exactly the same sentence embedding* as the process at top left.

Yes, this is wasteful computation, but that's about to change. The main thing to consider here is that if the matrices used were any matrix other than the identity matrix, changing the order of the words would change the resulting embedding. Let's see this in Python.

Introduction to automatic gradient computation (autograd)

Previously, you performed backpropagation by hand. Let's make it automatic!

In chapter 4, you learned about derivatives. Since then, you've been computing derivatives by hand for each neural network you train. Recall that this is done by moving backward through the neural network: first compute the gradient at the output of the network, then use that result to compute the derivative at the next-to-last component, and so on until all weights in the architecture have correct gradients. This logic for computing gradients can also be added to the tensor object. Let me show you what I mean. New code is in **bold**:

```
import numpy as np

class Tensor (object):

    def __init__(self, data, creators=None, creation_op=None):
        self.data = np.array(data)
        self.creation_op = creation_op
        self.creators = creators
        self.grad = None

    def backward(self, grad):
        self.grad = grad

        if(self.creation_op == "add"):
            self.creators[0].backward(grad)
            self.creators[1].backward(grad)

    def __add__(self, other):
        return Tensor(self.data + other.data,
                      creators=[self,other],
                      creation_op="add")

    def __repr__(self):
        return str(self.data.__repr__())

    def __str__(self):
        return str(self.data.__str__())

x = Tensor([1,2,3,4,5])
y = Tensor([2,2,2,2,2])

z = x + y
z.backward(Tensor(np.array([1,1,1,1,1])))
```

This method introduces two new concepts. First, each tensor gets two new attributes. `creators` is a list containing any tensors used in the creation of the current tensor (which defaults to `None`). Thus, when the two tensors `x` and `y` are added together, `z` has two

How to learn a framework

Oversimplified, frameworks are autograd + a list of prebuilt layers and optimizers.

You've been able to write (rather quickly) a variety of new layer types using the underlying autograd system, which makes it quite easy to piece together arbitrary layers of functionality. Truth be told, this is the main feature of modern frameworks, eliminating the need to handwrite each and every math operation for forward and backward propagation. Using frameworks greatly increases the speed with which you can go from idea to experiment and will reduce the number of bugs in your code.

Viewing a framework as merely an autograd system coupled with a big list of layers and optimizers will help you learn them. I expect you'll be able to pivot from this chapter into almost any framework fairly quickly, although the framework that's most similar to the API built here is PyTorch. Either way, for your reference, take a moment to peruse the lists of layers and optimizers in several of the big frameworks:

- https://pytorch.org/docs/stable/nn.html
- https://keras.io/layers/about-keras-layers
- https://www.tensorflow.org/api_docs/python/tf/layers

The general workflow for learning a new framework is to find the simplest possible code example, tweak it and get to know the autograd system's API, and then modify the code example piece by piece until you get to whatever experiment you care about.

```
def backward(self,grad=None, grad_origin=None):
    if(self.autograd):

        if(grad is None):
            grad = Tensor(np.ones_like(self.data))
```

One more thing before we move on. I'm adding a nice convenience function to `Tensor.backward()` that makes it so you don't have to pass in a gradient of 1s the first time you call `.backward()`. It's not, strictly speaking, necessary—but it's handy.

Let's see how to iterate using truncated backpropagation.

The following code shows truncated backpropagation in practice. Notice that it looks very similar to the iteration logic from chapter 13. The only real difference is that you generate a `batch_loss` at each step; and after every `bptt` steps, you backpropagate and perform a weight update. Then you keep reading through the dataset like nothing happened (even using the same hidden state from before, which only gets reset with each epoch):

```
def train(iterations=100):
    for iter in range(iterations):
        total_loss = 0
        n_loss = 0

        hidden = model.init_hidden(batch_size=batch_size)
        for batch_i in range(len(input_batches)):

            hidden = Tensor(hidden.data, autograd=True)
            loss = None
            losses = list()
            for t in range(bptt):
                input = Tensor(input_batches[batch_i][t], autograd=True)
                rnn_input = embed.forward(input=input)
                output, hidden = model.forward(input=rnn_input,
                                               hidden=hidden)
               target = Tensor(target_batches[batch_i][t], autograd=True)
                batch_loss = criterion.forward(output, target)
                losses.append(batch_loss)
                if(t == 0):
                    loss = batch_loss
                else:
                    loss = loss + batch_loss
            for loss in losses:
                ""
            loss.backward()
            optim.step()
            total_loss += loss.data
            log = "\r Iter:" + str(iter)
            log += " - Batch "+str(batch_i+1)+"/"+str(len(input_batches))
            log += " - Loss:" + str(np.exp(total_loss / (batch_i+1)))
            if(batch_i == 0):
                log += " - " + generate_sample(70,'\n').replace("\n"," ")
            if(batch_i % 10 == 0 or batch_i-1 == len(input_batches)):
                sys.stdout.write(log)
        optim.alpha *= 0.99
        print()
train()
```

```
Iter:0 - Batch 191/195 - Loss:148.00388828554404
Iter:1 - Batch 191/195 - Loss:20.588816924127116 mhnethet tttttt t t t
                                    ....
Iter:99 - Batch 61/195 - Loss:1.0533843281265225 I af the mands your
```

Secure aggregation

Let's average weight updates from zillions of people before anyone can see them.

The solution is to never let Bob put a gradient out in the open like that. How can Bob contribute his gradient if people shouldn't see it? The social sciences use an interesting technique called *randomized response*.

It goes like this. Let's say you're conducting a survey, and you want to ask 100 people whether they've committed a heinous crime. Of course, all would answer "No" even if you promised them you wouldn't tell. Instead, you have them flip a coin twice (somewhere you can't see), and tell them that if the first coin flip is heads, they should answer honestly; and if it's tails, they should answer "Yes" or "No" according to the second coin flip.

Given this scenario, you never actually ask people to tell you whether they committed crimes. The true answers are hidden in the random noise of the first and second coin flips. If 60% of people say "Yes," you can determine (using simple math) that about 70% of the people you surveyed committed heinous crimes (give or take a few percentage points). The idea is that the random noise makes it plausible that any information you learn about the person came from the noise instead of from them.

> **Privacy via plausible deniability**
>
> The level of chance that a particular answer came from random noise instead of an individual protects their privacy by giving them plausible deniability. This forms the basis for secure aggregation and, more generally, much of differential privacy.

You're looking only at aggregate statistics overall. (You never see anyone's answer directly; you see only pairs of answers or perhaps larger groupings.) Thus, the more people you can aggregate before adding noise, the less noise you have to add to hide them (and the more accurate the findings are).

In the context of federated learning, you could (if you wanted) add a ton of noise, but this would hurt training. Instead, first sum all the gradients from all the participants in such a way that no one can see anyone's gradient but their own. The class of problems for doing this is called *secure aggregation*, and in order to do it, you'll need one more (very cool) tool: *homomorphic encryption*.