PREV
3. Deep Learning from Scratch

NEXT
5. Convolutional Neural Networks

Chapter 4. Extensions

In the last chapter, after having spent two chapters reasoning from first principles about what deep learning models are and how they should work, we finally built our first deep learning model and trained it to solve the relatively simple problem of predicting house prices given numeric features about houses. On most real-world problems, however, successfully training deep learning models isn't so easy: while these models can conceivably find an optimal solution to any problem that can be framed as a supervised learning problem, in practice they often fail, and indeed there are few theoretical guarantees that a given model architecture will in fact find a good solution to a given problem. Still, there are some well-understood techniques that make neural network training more likely to succeed; these will be the focus of this chapter.

We'll start out by reviewing what neural networks are "trying to do" mathematically: find the minimum of a function. Then I'll show a series of techniques that can help the networks achieve this, demonstrating their effectiveness on the classic MNIST dataset of handwritten digits. We'll start with a loss function that is used throughout classification problems in deep learning, showing that it significantly accelerates learning (we've only covered regression problems thus far in this book because we hadn't yet introduced this loss function and thus haven't been able to do classification problems justice). On a similar note, we'll cover activation functions other than sigmoid and show why *they* might also accelerate learning, while discussing the trade-offs involved with activation functions in general. Next, we'll cover momentum, the most important (and straightforward) extension of the stochastic gradient descent optimization technique we've been using thus far, as well as briefly discussing what more advanced optimizers can do. We'll end by covering three techniques that are unrelated to each other but that are all essential: learning rate decay, weight initialization, and dropout. As we'll see, each of these techniques will help our neural network find successively more optimal solutions.

In the first chapter, we followed the "diagram-math-code" model for introducing each concept. Here, there isn't an obvious diagram for each technique, so we'll instead begin with the "intuition" for each technique, then follow up with the math (which will typically be much simpler than in the first chapter), and end with the code, which will really entail incorporating the technique into the framework we've built and thus describing precisely how it interacts with the

Find answers on the fly, or master something new. Subscribe today. See pricing options.

Some Intuition About Neural Networks

As we've seen, neural networks contain a bunch of weights; given these weights, along with some input data `X` and `y`, we can compute a resulting "loss." Figure 4-1 shows this extremely high-level (but still correct) view of neural networks.

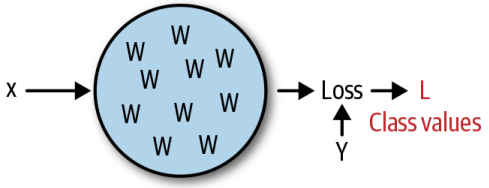

Figure 4-1. A simple way to think of a neural network with weights

In reality, each individual weight has some complex, nonlinear relationship with the features `X`, the target `y`, the other weights, and ultimately the loss `L`. If we plotted this out, varying the value of the weight while holding constant the values of the other weights, `X`, and `y`, and plotted the resulting value of the loss `L`, we could see something like what is shown in Figure 4-2.

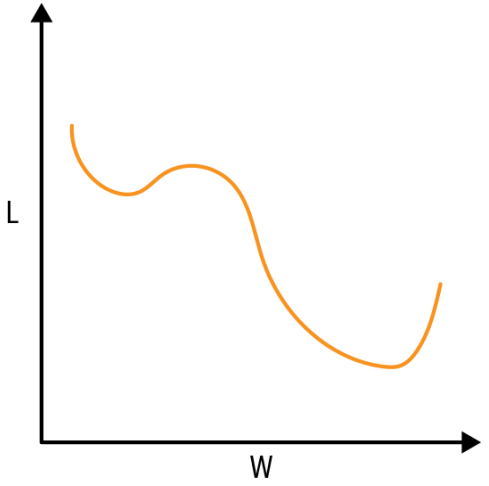

Figure 4-2. A neural network's weights versus its loss

When we start to train neural networks, we initialize each weight to have a value somewhere along the x-axis in Figure 4-2. Then, using the gradients we calculate during backpropagation, we iteratively update the weight, with our first update based on the slope of this curve at the initial value we happened to choose.[1] Figure 4-3 shows this geometric interpretation of what it means to update the weights in a neural network based on the gradients and the learning rate. The blue arrows on the left represent repeatedly applying this update rule with a smaller learning rate than the red arrows on the right; note that in both cases, the updates in the horizontal direction are proportional to the slope of the curve at the value of the weight (steeper slope means a larger update).

Find answers on the fly, or master something new. Subscribe today. See pricing options.

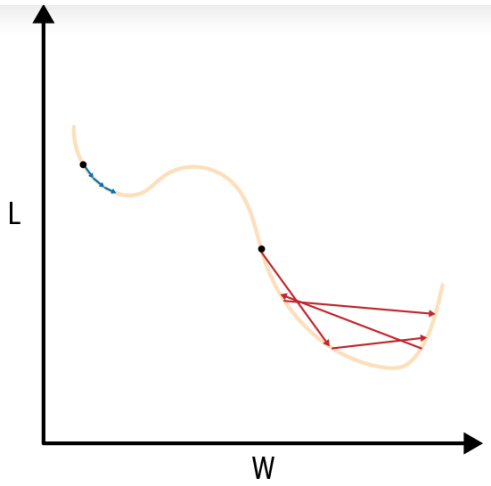

Figure 4-3. Updating the weights of a neural network as a function of the gradients and the learning rate, depicted geometrically

The goal of training a deep learning model is to move each weight to the "global" value for which the loss is minimized. As we can see from Figure 4-3, if the steps we take are too small, we risk ending up in a "local" minimum, which is less optimal than the global one (the path of a weight that follows that scenario is illustrated by the blue arrows). If the steps are too large, we risk "repeatedly hopping over" the global minimum, even if we are near it (this scenario is represented by the red arrows). This is the fundamental trade-off of tuning learning rates: if they are too small, we can get stuck in a local minimum; if they are too large, they can skip over the global minimum.

In reality, the picture is far more complicated than this. One reason is that there are thousands, if not millions, of weights in a neural network, so we are searching for a global minimum in a space that has thousands or millions of dimensions. Moreover, since we update the weights on each iteration as well as passing in a different `X` and `y`, *the curve we are trying to find the minimum of is constantly changing!* The latter is one of the main reasons neural networks were met with skepticism for so many years; it didn't seem like iteratively updating the weights in this way could actually find a globally desirable solution. Yann LeCun et al. say it best in a 2015 *Nature* article:

> *In particular, it was commonly thought that simple gradient descent would get trapped in poor local minima—weight configurations for which no small change would reduce the average error. In practice, poor local minima are rarely a problem with large networks. Regardless of the initial conditions, the system nearly always reaches solutions of very similar quality. Recent theoretical and empirical results strongly suggest that local minima are not a serious issue in general.*

So *in practice*, Figure 4-3 provides both a good mental model for why learning rates should not be too large or too small and adequate intuition for why many of the tricks we're going to learn in this chapter actually work. Equipped with this intuition of what neural networks are trying to do, let's start examining these tricks. We'll start with a loss function, the *softmax cross entropy* loss function, that works in large part because of its ability to provide steeper gradients to the weights than the mean squared error loss function we saw in the prior chapter.

The Softmax Cross Entropy Loss Function

Find answers on the fly, or master something new. Subscribe today. See pricing options.

`Loss` sent backward to the network `Layer`s and thus the greater would be all the gradients received by the parameters. It turns out that in classification problems, however, we can do better than this, since in such problems *we know that the values our network outputs should be interpreted as probabilities*; thus, not only should each value be between 0 and 1, but the vector of probabilities should sum to 1 for each observation we have fed through our network. The softmax cross entropy loss function exploits this to produce steeper gradients than the mean squared error loss for the same inputs. This function has two components: the first is the *softmax* function, and the second component is the "cross entropy" loss; we'll cover each of these in turn.

Component #1: The Softmax Function

For a classification problem with `N` possible classes, we'll have our neural network output a vector of `N` values for each observation. For a problem with three classes, these values could, for example, be:

```
[5, 3, 2]
```

MATH

Again, since this is a classification problem, we know that this *should* be interpreted as a vector of probabilities (the probability of this observation belonging to class 1, 2, or 3, respectively). One way to transform these values into a vector of probabilities would be to simply normalize them, summing and dividing by the sum:

$$\text{Normalize}\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}\right) = \begin{bmatrix} \frac{x_1}{x_1+x_2+x_3} \\ \frac{x_2}{x_1+x_2+x_3} \\ \frac{x_3}{x_1+x_2+x_3} \end{bmatrix}$$

However, there turns out to be a way that both produces steeper gradients and has some elegant mathematical properties: the softmax function. This function, for a vector of length 3, would be defined as:

$$\text{Softmax}\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}\right) = \begin{bmatrix} \frac{e^{x_1}}{e^{x_1}+e^{x_2}+e^{x_3}} \\ \frac{e^{x_2}}{e^{x_1}+e^{x_2}+e^{x_3}} \\ \frac{e^{x_3}}{e^{x_1}+e^{x_2}+e^{x_3}} \end{bmatrix}$$

INTUITION

The intuition behind the softmax function is that it more strongly amplifies the maximum value relative to the other values, forcing the neural network to be "less neutral" toward which prediction it thinks is the correct one in the context of a classification problem. Let's compare what both of these functions, normalize and softmax, would do to our preceding vector of probabilities:

```
normalize(np.array([5,3,2]))
```

```
array([0.5, 0.3, 0.2])
```

```
softmax(np.array([5,3,2]))
```

```
array([0.84, 0.11, 0.04])
```

We can see that the original maximum value—5—has a significantly higher value than it would have upon simply normalizing the data, and the other two values are lower than they were coming out of the `normalize` function. Thus, the `softmax` function is partway between normalizing the values and actually

Find answers on the fly, or master something new. Subscribe today. See pricing options.

Component #2: The Cross Entropy Loss

Recall that any loss function will take in a vector of probabilities $\begin{bmatrix} p_1 \\ \vdots \\ p_n \end{bmatrix}$ and a vector of actual values $\begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}$.

MATH

The cross entropy loss function, for each index `i` in these vectors, is:

$$\mathrm{CE}(p_i, y_i) = -y_i \times \log(p_i) - (1 - y_i) \times \log(1 - p_i)$$

INTUITION

To see why this makes sense as a loss function, consider that since every element of y is either 0 or 1, the preceding equation reduces to:

$$\mathrm{CE}(p_i, y_i) = \begin{cases} -log(1 - p_i) & \text{if } y_i = 0 \\ -log(p_i) & \text{if } y_i = 1 \end{cases}$$

Now we can break this down more easily. If $y = 0$, then the plot of the value of this loss versus the value of the mean squared error loss over the interval 0 to 1 is as depicted in Figure 4-4.

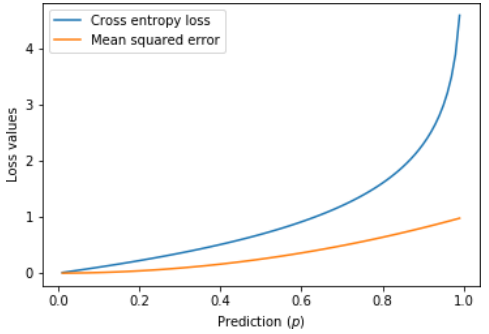

Figure 4-4. Cross entropy loss versus MSE when $y = 0$

Not only are the penalties for the cross entropy loss much higher over this interval,[2] but they get steeper at a higher rate; indeed, the value of the cross entropy loss approaches infinity as the difference between our prediction and the target approaches 1! The plot for when $y = 1$ is similar, just "flipped" (that is, it's rotated 180 degrees around the line $x = 0.5$).

So, for problems where we know the output will be between 0 and 1, the cross entropy loss produces steeper gradients than MSE. The real magic happens when we combine this loss with the softmax function—first feeding the neural network output through the softmax function to normalize it so the values add to 1, and then feeding the resulting probabilities into the cross entropy loss function.

Let's see what this looks like with the three-class scenario we've been using so far; the expression for the component of the loss vector from $i = 1$—that is, the first component of the loss for a given observation, which we'll denote as SCE_1—is:

$$SCE_1 = -y_1 \times log\left(\frac{e^{x_1}}{e^{x_1} + e^{x_2} + e^{x_3}}\right) - (1 - y_1) \times log\left(1 - \frac{e^{x_1}}{e^{x_1} + e^{x_2} + e^{x_3}}\right)$$

Find answers on the fly, or master something new. Subscribe today. See pricing options.

mathematically and easy to implement:

$$\frac{\partial SCE_1}{\partial x_1} = \frac{e^{x_1}}{e^{x_1} + e^{x_2} + e^{x_3}} - y_1$$

That means that the *total* gradient to the softmax cross entropy is:

$$\text{softmax}\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}\right) - \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

That's it! As promised, the resulting implementation is simple as well:

```
softmax_x = softmax(x, axis = 1)
loss_grad = softmax_x - y
```

Let's code this up.

CODE

To recap Chapter 3, any `Loss` class is expected to receive two 2D arrays, one with the network's predictions and the other with the targets. The number of rows in each array is the batch size, and the number of columns is the number of classes `n` in the classification problem; a row in each represents an observation in the dataset, with the `n` values in the row representing the neural network's best guess for the probabilities of that observation belonging to each of the `n` classes. Thus, we'll have to apply the `softmax` to *each row* in the `prediction` array. This leads to a first potential issue: we'll next feed the resulting numbers into the `log` function to compute the loss. This should worry you, since $log(x)$ goes to negative infinity as x goes to 0, and similarly, $1 - x$ goes to infinity as x goes to 1. To prevent extremely large loss values that could lead to numeric instability, we'll clip the output of the softmax function to be no less than 10^{-7} and no greater than 10^{7}.

Finally, we can put everything together!

```
class SoftmaxCrossEntropyLoss(Loss):
    def __init__(self, eps: float=1e-9)
        super().__init__()
        self.eps = eps
        self.single_output = False

    def _output(self) -> float:

        # applying the softmax function to each row (observation)
        softmax_preds = softmax(self.prediction, axis=1)

        # clipping the softmax output to prevent numeric instability
        self.softmax_preds = np.clip(softmax_preds, self.eps, 1 - sel

        # actual loss computation
softmax_cross_entropy_loss = (
    -1.0 * self.target * np.log(self.softmax_preds) - \
        (1.0 - self.target) * np.log(1 - self.softmax_preds)
)

        return np.sum(softmax_cross_entropy_loss)

    def _input_grad(self) -> ndarray:

        return self.softmax_preds - self.target
```

Soon I'll show via some experiments on the MNIST dataset how this loss is an improvement on the mean squared error loss. But first let's discuss the trade-offs involved with choosing an activation function and see if there's a better choice than sigmoid.

Find answers on the fly, or master something new. Subscribe today. See pricing options.

We argued in Chapter 2 that sigmoid was a good activation function because it:

- Was a nonlinear and monotonic function
- Provided a "regularizing" effect on the model, forcing the intermediate features down to a finite range, specifically between 0 and 1

Nevertheless, sigmoid has a downside, similar to the downside of the mean squared error loss: *it produces relatively flat gradients* during the backward pass. The gradient that gets passed to the sigmoid function (or any function) on the backward pass represents how much the function's *output* ultimately affects the loss; because the maximum slope of the sigmoid function is 0.25, these gradients will *at best* be divided by 4 when sent backward to the previous operation in the model. Worse still, when the input to the sigmoid function is less than –2 or greater than 2, the gradient those inputs receive will be almost 0, since *sigmoid*(x) is almost flat at $x = -2$ or $x = 2$. What this means is that any parameters influencing these inputs will receive small gradients, and our network could learn slowly as a result.[3] Furthermore, if multiple sigmoid activation functions are used in successive layers of a neural network, this problem will compound, further diminishing the gradients that weights earlier in the neural network could receive.

What would an activation "at the other extreme"—one with the opposite strengths and weaknesses—look like?

THE OTHER EXTREME: THE RECTIFIED LINEAR UNIT

The Rectified Linear Unit, or ReLU, activation is a commonly used activation with the opposite strengths and weaknesses of sigmoid. ReLU is simply defined to be 0 if x is less than 0, and x otherwise. A plot of this is shown in Figure 4-5.

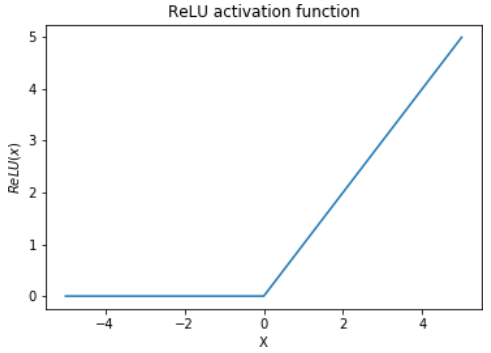

Figure 4-5. ReLU activation

This is a "valid" activation function in the sense that it is monotonic and nonlinear. It produces much larger gradients than sigmoid—1 if the input to the function is greater than 0, and 0 otherwise, for an average of 0.5—whereas the *maximum* gradient sigmoid can produce is 0.25. ReLU activation is a very popular choice in deep neural network architectures, because its downside (that it draws a sharp, somewhat arbitrary distinction between values less than or greater than 0) can be addressed by other techniques, including some that will be covered in this chapter, and its benefits (of producing large gradients) are critical to training the weights in the architectures of deep neural networks.

Nevertheless, there's an activation function that is a happy medium between these two, and that we'll use in the demos in this chapter: Tanh.

A HAPPY MEDIUM: TANH

Find answers on the fly, or master something new. Subscribe today. See pricing options.

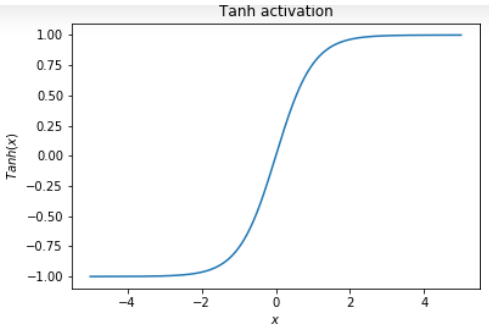

Figure 4-6. Tanh activation

This function produces significantly steeper gradients than sigmoid; specifically, the maximum gradient of Tanh turns out to be 1, in contrast to sigmoid's 0.25. Figure 4-7 shows the gradients of these two functions.

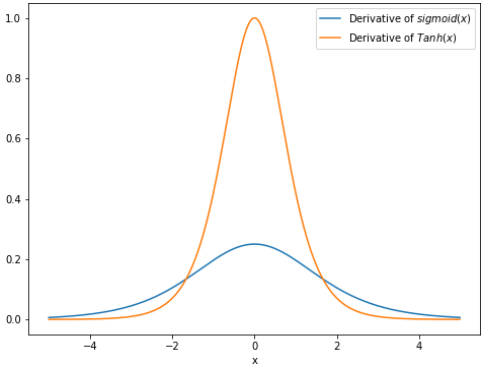

Figure 4-7. Sigmoid derivative versus Tanh derivative

In addition, just as $f(x) = sigmoid(x)$ has the easy-to-express derivative $f'(x) = sigmoid(x) \times (1 - sigmoid(x))$, so too does $f(x) = tanh(x)$ have the easy-to-express derivative $f'(x) = (1 - tanh(x))^2$.

The point here is that there are trade-offs involved with choosing an activation function regardless of the architecture: we want an activation function that will allow our network to learn nonlinear relationships between input and output, while not adding unnecessary complexity that will make it harder for the network to find a good solution. For example, the "Leaky ReLU" activation function allows a slight negative slope when the input to the ReLU function is less than 0, enhancing ReLU's ability to send gradients backward, and the "ReLU6" activation function caps the positive end of ReLU at 6, introducing even more nonlinearity into the network. Still, both of these activation functions are more complex than ReLU; and if the problem we are dealing with is relatively simple, those more sophisticated activation functions could make it even harder for the network to learn. Thus, in the models we demo in the rest of this book, we'll simply use the Tanh activation function, which balances these considerations well.

Now that we've chosen an activation function, let's use it to run some experiments.

Find answers on the fly, or master something new. Subscribe today. See pricing options.

We've justified using `Tanh` throughout our experiments, so let's get back to the original point of this section: showing why the softmax cross entropy loss is so pervasive throughout deep learning.[4] We'll use the MNIST dataset, which consists of black-and-white images of handwritten digits that are 28×28 pixels, with the value of each pixel ranging from 0 (white) to 255 (black). Furthermore, the dataset is predivided into a training set of 60,000 images and a testing set of 10,000 additional images. In the book's GitHub repo, we show a helper function to read both the images and their corresponding labels into training and testing sets using the following line of code:

```
X_train, y_train, X_test, y_test = mnist.load()
```

Our goal will be to train a neural network to learn which of the 10 digits from 0 to 9 the image contains.

Data Preprocessing

For classification, we have to perform *one-hot encoding* to transform our vectors representing the labels into an `ndarray` of the same shape as the predictions: specifically, we'll map the label "0" to a vector with a 1 in the first position (at index 0) and 0s in all the other positions, "1" to a vector with a 1 in the second position (at index 1), and so on:

$$[0, 2, 1] \Rightarrow \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \end{bmatrix}$$

Finally, it is always helpful to scale our data to mean 0 and variance 1, just as we did with the "real-world" datasets in the prior chapters. Here, however, since each data point is an image, we won't scale each *feature* to have mean 0 and variance 1, since that would result in the values of adjacent pixels being changed by different amounts, which could distort the image! Instead, we'll simply provide a global scaling to our dataset that subtracts off the overall mean and divides by the overall variance (note that we use the statistics from the training set to scale the testing set):

```
X_train, X_test = X_train - np.mean(X_train), X_test - np.mean(X_train
X_train, X_test = X_train / np.std(X_train), X_test / np.std(X_train)
```

Model

We'll have to define our model to have 10 outputs for each input: one for each of the probabilities of our model belonging to each of the 10 classes. Since we know each of our outputs will be a probability, we'll give our model a `sigmoid` activation on the last layer. Throughout this chapter, to illustrate whether the "training tricks" we're describing actually enhance our models' ability to learn, we'll use a consistent model architecture of a two-layer neural network with the number of neurons in the hidden layer close to the geometric mean of our number of inputs (784) and our number of outputs (10): $89 \approx \sqrt{784 \times 10}$.

Let's now turn to our first experiment, comparing a neural network trained with simple mean squared error loss to one trained with softmax cross entropy loss. The loss values you see displayed are per observation (recall that on average cross entropy loss will have absolute loss values three times as large as mean squared error loss). If we run:

```
model = NeuralNetwork(
    layers=[Dense(neurons=89,
                  activation=Tanh()),
            Dense(neurons=10,
                  activation=Sigmoid())],
```

Find answers on the fly, or master something new. Subscribe today. See pricing options.

```
optimizer = SGD(0.1)

trainer = Trainer(model, optimizer)
trainer.fit(X_train, train_labels, X_test, test_labels,
            epochs = 50,
            eval_every = 10,
            seed=20190119,
            batch_size=60);

calc_accuracy_model(model, X_test)
```

it gives us:

```
Validation loss after 10 epochs is 0.611
Validation loss after 20 epochs is 0.428
Validation loss after 30 epochs is 0.389
Validation loss after 40 epochs is 0.374
Validation loss after 50 epochs is 0.366

The model validation accuracy is: 72.58%
```

Now let's test the claim we made earlier in the chapter: that the softmax cross entropy loss function would help our model learn faster.

Experiment: Softmax Cross Entropy Loss

First let's change the preceding model to:

```
model = NeuralNetwork(
    layers=[Dense(neurons=89,
                  activation=Tanh()),
            Dense(neurons=10,
                  activation=Linear())],
            loss = SoftmaxCrossEntropy(),
seed=20190119)
```

> **NOTE**
>
> Since we are now feeding the model outputs through the softmax function as part of the loss, we no longer need to feed them through a sigmoid activation function.

Then we run it for model for 50 epochs, which gives us these results:

```
Validation loss after 10 epochs is 0.630
Validation loss after 20 epochs is 0.574
Validation loss after 30 epochs is 0.549
Validation loss after 40 epochs is 0.546
Loss increased after epoch 50, final loss was 0.546, using the mo
epoch 40

The model validation accuracy is: 91.01%
```

Indeed, changing our loss function to one that gives much steeper gradients alone gives a huge boost to the accuracy of our model![5]

Of course, we can do significantly better than this, even without changing our architecture. In the next section, we'll cover momentum, the most important and straightforward extension to the stochastic gradient descent optimization technique we've been using up until now.

Find answers on the fly, or master something new. Subscribe today. See pricing options.

So far, we've been using only one "update rule" for our weights at each time step. Simply take the derivative of the loss with respect to the weights and move the weights in the resulting correct direction. This means that our `_update_rule` function in the `Optimizer` looked like:

```
update = self.lr*kwargs['grad']
kwargs['param'] -= update
```

Let's first cover the intuition for why we might want to extend this update rule to incorporate momentum.

Intuition for Momentum

Recall Figure 4-3, which plotted an individual parameter's value against the loss value from the network. Imagine a scenario in which the parameter's value is continually updated in the same direction because the loss continues to decrease with each iteration. This would be analogous to the parameter "rolling down a hill," and the value of the update at each time step would be analogous to the parameter's "velocity." In the real world, however, objects don't instantaneously stop and change directions; that's because they have *momentum*, which is just a concise way of saying their velocity at a given instant is a function not just of the forces acting on them in that instant but also of their accumulated past velocities, with more recent velocities weighted more heavily. This physical interpretation is the motivation for applying momentum to our weight updates. In the next section we'll make this precise.

Implementing Momentum in the Optimizer Class

Basing our parameter updates on momentum means that *the parameter update at each time step will be a weighted average of the parameter updates at past time steps, with the weights decayed exponentially*. There will thus be a second parameter we have to choose, the momentum parameter, which will determine the degree of this decay; the higher it is, the more the weight update at each time step will be based on the parameter's accumulated momentum as opposed to its current velocity.

MATH

Mathematically, if our momentum parameter is μ, and the gradient at each time step is ∇_t, our weight update is:

$$\text{update} = \nabla_t + \mu \times \nabla_{t-1} + \mu^2 \times \nabla_{t-2} + ...$$

If our momentum parameter was 0.9, for example, we would multiply the gradient from one time step ago by 0.9, the one from two time steps ago by $0.9^2 = 0.81$, the one from three time steps ago by $0.9^3 = 0.729$, and so on, and then finally add all of these to the gradient from the current time step to get the overall weight update for the current time step.

CODE

How do we implement this? Do we have to compute an infinite sum every time we want to update our weights?

It turns out there is a cleverer way. Our `Optimizer` will keep track of a separate quantity representing the *history* of parameter updates in addition to just receiving a gradient at each time step. Then, at each time step, we'll use the current gradient to update this history and compute the actual parameter update as a function of this history. Since momentum is loosely based on an analogy with physics, we'll call this quantity "velocity."

How should we update velocity? It turns out we can use the following steps:

Find answers on the fly, or master something new. Subscribe today. See pricing options.

2. Add the gradient.

This results in the velocity taking on the following values at each time step, starting at $t = 1$:

1. ∇_1
2. $\nabla_2 + \mu \times \nabla_1$
3. $\nabla_3 + \mu \times (\nabla_2 + \mu \times \nabla_1) = \mu \times \nabla_2 + \mu^2 \times \nabla_1)$

With this, we can use the velocity as the parameter update! We can then incorporate this into a new subclass of `Optimizer` that we'll call `SGDMomentum`; this class will have `step` and `_update_rule` functions that look as follows:

```
def step(self) -> None:
    '''
    If first iteration: intialize "velocities" for each param.

    Otherwise, simply apply _update_rule.
    '''
    if self.first:
        # now we will set up velocities on the first iteration
        self.velocities = [np.zeros_like(param)
                           for param in self.net.params()]
        self.first = False

    for (param, param_grad, velocity) in zip(self.net.params(),
                                             self.net.param_grads()
                                             self.velocities):
        # pass in velocity into the "_update_rule" function
        self._update_rule(param=param,
                          grad=param_grad,
                          velocity=velocity)

def _update_rule(self, **kwargs) -> None:
        '''
        Update rule for SGD with momentum.
        '''
        # Update velocity
        kwargs['velocity'] *= self.momentum
        kwargs['velocity'] += self.lr * kwargs['grad']

        # Use this to update parameters
        kwargs['param'] -= kwargs['velocity']
```

Let's see if this new optimizer can improve our network's training.

Experiment: Stochastic Gradient Descent with Momentum

Let's train the same neural network with one hidden layer on the MNIST dataset, changing nothing except using `optimizer = SGDMomentum(lr=0.1, momentum=0.9)` as the optimizer instead of `optimizer = SGD(lr=0.1)`:

```
Validation loss after 10 epochs is 0.441
Validation loss after 20 epochs is 0.351
Validation loss after 30 epochs is 0.345
Validation loss after 40 epochs is 0.338
Loss increased after epoch 50, final loss was 0.338, using the mo

The model validation accuracy is: 95.51%
```

You can see that the loss is significantly lower and the accuracy significantly higher, which is simply a result of adding momentum into our parameter update rule![6]

Find answers on the fly, or master something new. Subscribe today. See pricing options.

rate, we can also automatically decay the learning rate as training proceeds using some rule. The most common such rules are covered next.

Learning Rate Decay

> *[The learning rate] is often the single most important hyper-parameter and one should always make sure that it has been tuned.*
>
> —Yoshua Bengio, *Practical Recommendations for Gradient-Based Training of Deep Architectures*, 2012

The motivation for decaying the learning rate as training progresses comes, yet again, from Figure 4-3 in the previous section: while we want to "take big steps" toward the beginning of training, it is likely that as we continue to iteratively update the weights, we will reach a point where we start to "skip over" the minimum. Note that this won't *necessarily* be a problem, since if the relationship between our weights and the loss "smoothly declines" as we approach the minimum, as in Figure 4-3, the magnitude of the gradients will automatically decrease as the slope decreases. Still, this may not happen, and even if it does, learning rate decay can give us more fine-grained control over this process.

Types of Learning Rate Decay

There are different ways of decaying the learning rate. The simplest is linear decay, where the learning rate declines linearly from its initial value to some terminal value, with the actual decline being implemented at the end of each epoch. More precisely, at time step *t*, if the learning rate we want to start with is α_{start}, and our final learning rate is α_{end}, then our learning rate at each time step is:

$$\alpha_t = \alpha_{start} - (\alpha_{start} - \alpha_{end}) \times \frac{t}{N}$$

where *N* is the total number of epochs.

Another simple method that works about as well is exponential decay, in which the learning rate declines by a constant *proportion* each epoch. The formula here would simply be:

$$\alpha_t = \alpha \times \delta^t$$

where:

$$\delta = \frac{\alpha_{end}}{\alpha_{start}}^{\frac{1}{N-1}}$$

Implementing these is straightforward: we'll initialize our `Optimizer`s to have a "final learning rate" `final_lr` that the initial learning rate will decay toward throughout the epochs of training:

```
    def __init__(self,
                 lr: float = 0.01,
                 final_lr: float = 0,
                 decay_type: str = 'exponential')
        self.lr = lr
        self.final_lr = final_lr
        self.decay_type = decay_type
```

Then, at the beginning of training, we can call a `_setup_decay` function that calculates how much the learning rate will decay at each epoch:

```
    self.optim._setup_decay()
```

These calculations will implement the linear and exponential learning rate decay formulas we just saw:

Find answers on the fly, or master something new. Subscribe today. See pricing options.

```
        if not self.decay_type:
            return
        elif self.decay_type == 'exponential':
            self.decay_per_epoch = np.power(self.final_lr / self.lr,
                                            1.0 / (self.max_epochs-1))
        elif self.decay_type == 'linear':
            self.decay_per_epoch = (self.lr - self.final_lr) / (self.max_
```

Then, at the end of each epoch, we'll actually decay the learning rate:

```
    def _decay_lr(self) -> None:

        if not self.decay_type:
            return

        if self.decay_type == 'exponential':
            self.lr *= self.decay_per_epoch

        elif self.decay_type == 'linear':
            self.lr -= self.decay_per_epoch
```

Finally, we'll call the `_decay_lr` function from the `Trainer` during the `fit` function, at the end of each epoch:

```
    if self.optim.final_lr:
        self.optim._decay_lr()
```

Let's run some experiments to see if this improves training.

Experiments: Learning Rate Decay

Next we try training the same model architecture with learning rate decay. We initialize the learning rates so that the "average learning rate" over the run is equal to the previous learning rate of 0.1: for the run with linear learning rate decay, we initialize the learning rate to 0.15 and decay it down to 0.05, and for the run with exponential decay, we initialize the learning rate to 0.2 and decay it down to 0.05. For the linear decay run with:

```
    optimizer = SGDMomentum(0.15, momentum=0.9, final_lr=0.05, decay_type=
```

we have:

```
    Validation loss after 10 epochs is 0.403
    Validation loss after 20 epochs is 0.343
    Validation loss after 30 epochs is 0.282
    Loss increased after epoch 40, final loss was 0.282, using the mo
    The model validation accuracy is: 95.91%
```

For the run with exponential decay, with:

```
    optimizer = SGDMomentum(0.2, momentum=0.9, final_lr=0.05, decay_type='
```

we have:

```
    Validation loss after 10 epochs is 0.461
    Validation loss after 20 epochs is 0.323
    Validation loss after 30 epochs is 0.284
    Loss increased after epoch 40, final loss was 0.284, using the mo
    The model validation accuracy is: 96.06%
```

Find answers on the fly, or master something new. Subscribe today. See pricing options.

Next up: how and why to more intelligently initialize the weights of our model.

Weight Initialization

As we mentioned in the section on activation functions, several activation functions, such as sigmoid and Tanh, have their steepest gradients when their inputs are 0, with the functions quickly flattening out as the inputs move away from 0. This can potentially limit the effectiveness of these functions, because if many of the inputs have values far from 0, the weights attached to those inputs will receive very small gradients on the backward pass.

This turns out to be a major problem in the neural networks we are now dealing with. Consider the hidden layer in the MNIST network we've been looking at. This layer will receive 784 inputs and then multiply them by a weight matrix, ending up with some number `n` of neurons (and then optionally add a bias to each neuron). Figure 4-8 shows the distribution of these `n` values in the hidden layer of our neural network (with 784 inputs) before and after feeding them through the `Tanh` activation function.

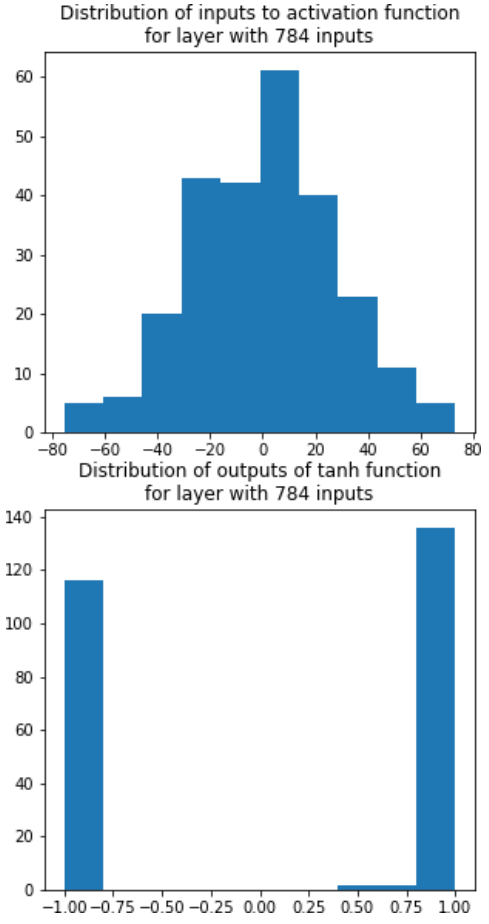

Figure 4-8. Distribution of inputs to activation function and activations

After being fed through the activation function, most of the activations are either

Find answers on the fly, or master something new. Subscribe today. See pricing options.

Since we initialized each weight to have variance 1 ($\text{Var}(w_{i,j}) = 1$—and $\text{Var}(b_n) = 1$) and $\text{Var}(X_1 + X_2) = \text{Var}(X_1) + \text{Var}(X_2)$ for independent random variables X_1 and X_2, we have:

$$\text{Var}(f_n) = 785$$

That gives it a standard deviation ($\sqrt{785}$) of just over 28, which reflects the spread of the values we see in the top half of Figure 4-8.

This tells us we have a problem. But is the problem simply that the features that we feed into the activation functions can't be "too spread out"? If that were the problem, we could simply divide the features by some value to reduce their variance. However, that invites an obvious question: how do we know what to divide the values by? The answer is that the values should be scaled *based on the number of neurons being fed into the layer*. If we had a multilayer neural network, and one layer had 200 neurons and the next layer had 100, the 200-neuron layer would pass values forward that had a wider distribution than the 100-neuron layer. This is undesirable—we don't want the *scale* of the features our neural network learns during training to depend on the number of features passed forward, for the same reason that we don't want our network's predictions to depend on the scale of our input features. Our model's predictions shouldn't be affected, for example, if we multiplied or divided all the values in a feature by 2.

There are several ways to correct for this; here we'll cover the single most prevalent one, suggested by the prior paragraph: we can *adjust the initial variance of the weights based on the number of neurons in the layers they connect so that the values passed forward to the following layer during the forward pass* and *backward to the prior layer during the backward pass* have roughly the same scale. Yes, we have to think about the backward pass too, since the same problem exists there: the variance of the gradients the layer receives during backpropagation will depend directly on the number of features in the *following* layer since that is the one that sends gradients backward to the layer in question.

Math and Code

How, specifically, do we balance these concerns? If each layer has n_{in} neurons feeding in and n_{out} neurons coming out, the variance for each weight that would keep the variance of the resulting features constant *just on the forward pass* would be:

$$\frac{1}{n_{in}}$$

Similarly, the weight variance that would keep the variance of the features constant on the backward pass would be:

$$\frac{1}{n_{out}}$$

As a compromise between these, what is most often called *Glorot initialization*[7] involves initializing the variance of the weights in each layer to be:

$$\frac{2}{n_{in} + n_{out}}$$

Coding this up is simple—we add a `weight_init` argument to each layer, and we add the following to our `_setup_layer` function:

```
if self.weight_init == "glorot":
    scale = 2/(num_in + self.neurons)
else:
    scale = 1.0
```

Now our models will look like:

Find answers on the fly, or master something new. Subscribe today. See pricing options.

```
                    activation=Tanh(),
                    weight_init="glorot"),
                Dense(neurons=10,
                    activation=Linear(),
                    weight_init="glorot")],
                loss = SoftmaxCrossEntropy(),
    seed=20190119)
```

with `weight_init="glorot"` specified for each layer.

Experiments: Weight Initialization

Running the same models from the prior section but with the weights initialized using Glorot initialization gives:

```
Validation loss after 10 epochs is 0.352
Validation loss after 20 epochs is 0.280
Validation loss after 30 epochs is 0.244
Loss increased after epoch 40, final loss was 0.244, using the mo
The model validation accuracy is: 96.71%
```

for the model with linear learning rate decay, and:

```
Validation loss after 10 epochs is 0.305
Validation loss after 20 epochs is 0.264
Validation loss after 30 epochs is 0.245
Loss increased after epoch 40, final loss was 0.245, using the mo
The model validation accuracy is: 96.71%
```

for the model with exponential learning rate decay. We see another significant drop in the loss here, from the 0.282 and 0.284 we achieved earlier down to 0.244 and 0.245! Note that with all of these changes, *we haven't been increasing the size or training time of our model*; we've simply tweaked the training process based on our intuition about what neural networks are trying to do that I showed at the beginning of this chapter.

There is one last technique we'll cover in this chapter. As motivation, you may have noticed that *none of the models we've used throughout this chapter were deep learning models*; rather, they were simply neural networks with one hidden layer. That is because without *dropout*, the technique we will now learn, deep learning models are very challenging to train effectively without overfitting.

Dropout

In this chapter, I've shown several modifications to our neural network's training procedure that got it closer and closer to its global minimum. You may have noticed that we haven't tried the seemingly most obvious thing: adding more layers to our network or more neurons per layer. The reason is that simply adding more "firepower" to most neural network architectures can make it *more* difficult for the network to find a solution that generalizes well. The intuition here is that, while adding more capacity to a neural network allows it to model more complex relationships between input and output, it also risks leading the network to a solution that is overfit to the training data. *Dropout allows us to add capacity to our neural networks while in most cases making it less likely that the network will overfit*.

What specifically *is* dropout?

Definition

Dropout simply involves randomly choosing some proportion p of the neurons in a layer and setting them equal to 0 during each forward pass of training. This odd

Find answers on the fly, or master something new. Subscribe today. See pricing options.

networks, where the features being learned are, by construction, multiple layers of abstraction removed from the original features.

Though dropout can help our network avoid overfitting during training, we still want to give our network its "best shot" of making correct predictions when it comes time to predict. So, the `Dropout` operation will have two "modes": a "training" mode in which dropout is applied, and an "inference" mode, in which it is not. This causes another problem, however: applying dropout to a layer reduces the overall magnitude of the values being passed forward by a factor of $1 - p$ on average, meaning that if the weights in the following layers would normally expect values with magnitude M, they are instead getting magnitude $M \times (1 - p)$. We want to mimic this magnitude shift when running the network in inference mode, so, in addition to removing dropout, we'll multiply *all* the values by $1 - p$.

To make this more clear, let's code it up.

Implementation

We can implement dropout as an `Operation` that we'll tack onto the end of each layer. It will look as follows:

```
class Dropout(Operation):

    def __init__(self,
                 keep_prob: float = 0.8):
        super().__init__()
        self.keep_prob = keep_prob

    def _output(self, inference: bool) -> ndarray:
        if inference:
            return self.inputs * self.keep_prob
        else:
            self.mask = np.random.binomial(1, self.keep_prob,
                                           size=self.inputs.shape)
            return self.inputs * self.mask

    def _input_grad(self, output_grad: ndarray) -> ndarray:
        return output_grad * self.mask
```

On the forward pass, when applying dropout, we save a "mask" representing the individual neurons that got set to 0. Then, on the backward pass, we multiply the gradient the operation receives by this mask. This is because dropout makes the gradient 0 for the input values that are zeroed out (since changing their values will now have no effect on the loss) and leaves the other gradients unchanged.

ADJUSTING THE REST OF OUR FRAMEWORK TO ACCOMMODATE DROPOUT

You may have noticed that we included an `inference` flag in the `_output` method that affects whether dropout is applied or not. For this flag to be called properly, we actually have to add it in several other places throughout training:

1. The `Layer` and `NeuralNetwork` `forward` methods will take in `inference` as an argument (`False` by default) and pass the flag into each `Operation`, so that every `Operation` will behave differently in training mode than in inference mode.

2. Recall that in the `Trainer`, we evaluate the trained model on the testing set every `eval_every` epochs. Now, every time we do that, we'll evaluate with the `inference` flag equal to `True`:

```
test_preds = self.net.forward(X_test, inference=True)
```

Find answers on the fly, or master something new. Subscribe today. See pricing options.

```
    def __init__(self,
                 neurons: int,
                 activation: Operation = Linear(),
                 dropout: float = 1.0,
                 weight_init: str = "standard")
```

and we append the `dropout` operation by adding the following to the class's `_setup_layer` function:

```
    if self.dropout < 1.0:
        self.operations.append(Dropout(self.dropout))
```

That's it! Let's see if dropout works.

Experiments: Dropout

First, we see that adding dropout to our existing model does indeed decrease the loss. Adding dropout of 0.8 (so that 20% of the neurons get set to 0) to the first layer, so that our model looks like:

```
mnist_soft = NeuralNetwork(
    layers=[Dense(neurons=89,
                  activation=Tanh(),
                  weight_init="glorot",
                  dropout=0.8),
            Dense(neurons=10,
                  activation=Linear(),
                  weight_init="glorot")],
            loss = SoftmaxCrossEntropy(),
seed=20190119)
```

and training the model with the same hyperparameters as before (exponential weight decay from an initial learning rate of 0.2 to a final learning rate of 0.05) results in:

```
Validation loss after 10 epochs is 0.285
Validation loss after 20 epochs is 0.232
Validation loss after 30 epochs is 0.199
Validation loss after 40 epochs is 0.196
Loss increased after epoch 50, final loss was 0.196, using the mo
The model validation accuracy is: 96.95%
```

This is another significant decrease in loss over what we saw previously: the model achieves a minimum loss of 0.196, compared to 0.244 before.

The real power of dropout comes when we add more layers. Let's change the model we've been using throughout this chapter to be a deep learning model, defining the first hidden layer to have twice as many neurons as did our hidden layer before (178) and our second hidden layer to have half as many (46). Our model looks like:

```
model = NeuralNetwork(
    layers=[Dense(neurons=178,
                  activation=Tanh(),
                  weight_init="glorot",
                  dropout=0.8),
            Dense(neurons=46,
                  activation=Tanh(),
                  weight_init="glorot",
                  dropout=0.8),
            Dense(neurons=10,
                  activation=Linear(),
                  weight_init="glorot")],
            loss = SoftmaxCrossEntropy(),
seed=20190119)
```

Find answers on the fly, or master something new. Subscribe today. See pricing options.

Training this model with the same optimizer as before yields another significant decrease in the minimum loss achieved—and an increase in accuracy!

```
Validation loss after 10 epochs is 0.321
Validation loss after 20 epochs is 0.268
Validation loss after 30 epochs is 0.248
Validation loss after 40 epochs is 0.222
Validation loss after 50 epochs is 0.217
Validation loss after 60 epochs is 0.194
Validation loss after 70 epochs is 0.191
Validation loss after 80 epochs is 0.190
Validation loss after 90 epochs is 0.182
Loss increased after epoch 100, final loss was 0.182, using the m
The model validation accuracy is: 97.15%
```

More importantly, however, this improvement isn't possible without dropout. Here are the results of training the same model with no dropout:

```
Validation loss after 10 epochs is 0.375
Validation loss after 20 epochs is 0.305
Validation loss after 30 epochs is 0.262
Validation loss after 40 epochs is 0.246
Loss increased after epoch 50, final loss was 0.246, using the mo
The model validation accuracy is: 96.52%
```

Without dropout, the deep learning model performs *worse* than the model with just one hidden layer—despite having more than twice as many parameters and taking more than twice as long to train! This illustrates how essential dropout is for training deep learning models effectively; indeed, dropout was an essential component of the ImageNet-winning model from 2012 that kicked off the modern deep learning era.[8] Without dropout, you might not be reading this book!

Conclusion

In this chapter, you have learned some of the most common techniques for improving neural network training, learning both the intuition for why they work as well as the low-level details of how they work. To summarize these, we'll leave you with a checklist of things you can try to squeeze some extra performance out of your neural network, regardless of the domain:

- Add momentum—or one of many similarly effective advanced optimization techniques—to your weight update rule.
- Decay your learning rate over time using either linear or exponential decay as shown in this chapter or a more modern technique such as cosine decay. In fact, more effective learning rate schedules vary the learning rate based not just on each epoch but also *on the loss on the testing set*, decreasing the learning rate only when this loss fails to decrease. You should try implementing this as an exercise!
- Ensure the scale of your weight initialization is a function of the number of neurons in your layer (this is done by default in most neural network libraries).
- Add dropout, especially if your network contains multiple fully connected layers in succession.

Next, we'll shift to discussing advanced architectures specialized for specific domains, starting with convolutional neural networks, which are specialized to understand image data. Onward!

1 In addition, as we saw in Chapter 3, we multiply these gradients by a learning rate to give us more fine-grained control over how much the

Find answers on the fly, or master something new. Subscribe today. See pricing options.

2 We can be more specific: the average value of $-log(1-x)$ over the interval 0 to 1 turns out to be 1, while the average value of $x \quad 2 \quad$ over the same interval is just $\frac{1}{3}$.

3 To get an intuition for why this can happen: imagine a weight w is contributing to a feature f (so that $f = w \times x_1 + \ldots$), and during the forward pass of our neural network, $f = -10$ for some observation. Because $sigmoid(x)$ is so flat at $x = -10$, changing the value of w will have almost no effect on the model prediction and thus the loss.

4 For example, TensorFlow's MNIST classification tutorial uses the `softmax_cross_entropy_with_logits` function, and PyTorch's `nn.CrossEntropyLoss` actually computes the softmax function inside it.

5 You may argue that the softmax cross entropy loss is getting an "unfair advantage" here, since the softmax function normalizes the values it receives so that they add to 1, whereas the mean squared error loss simply gets 10 inputs that have been fed through the `sigmoid` function and have not been normalized to add to 1. However, on the book's website I show that MSE still performs worse than the softmax cross entropy loss even after normalizing the inputs to the mean squared error loss so that they sum to 1 for each observation.

6 Moreover, momentum is just one way we can use information beyond the gradient from the current batch of data to update the parameters; we briefly cover other update rules in Appendix A, and you can see these update rules implemented in the Lincoln library included on the book's GitHub repo.

7 This is so known because it was proposed by Glorot and Bengio in a 2010 paper: "Understanding the difficulty of training deep feedforward neural networks" (http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf).

8 For more on this, see G. E. Hinton et al., "Improving neural networks by preventing co-adaptation of feature detectors".

Support / Sign Out

PREV
3. Deep Learning from Scratch

NEXT
5. Convolutional Neural Networks

Find answers on the fly, or master something new. Subscribe today. See pricing options.