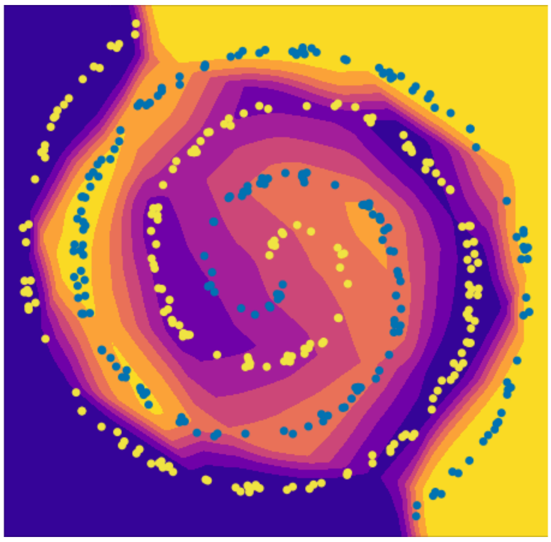


10 Days Of Grad: Deep Learning From The First Principles (/neural-networks)

Day 2: What Do Hidden Layers Do?

Bogdan Penkovsky

Feb 4, 2019 · 21 min read · 10 Days Of Grad (http://penkovsky.com/categories/10-days-of-grad/)

In the previous article (/neural-networks/day1/), we have introduced the concept of learning in a single-layer neural network. Today, we will learn about the benefits of multi-layer neural networks, how to properly design and train them.

Sometimes I discuss neural networks with students who have just started discovering machine learning techniques:

"I have built a handwritten digits (https://en.wikipedia.org/wiki/MNIST_database) recognition network. But my accuracy is only **Y**."

"*It seems to be much less than state-of-the-art*", I contemplate.

"Indeed. Maybe the reason is **X**?"

Usually **X** is not the reason. The real cause appears to be much more trivial: instead of a multi-layer neural network the student has built a single-layer neural network or its equivalent. This network acts as a *linear classifier (https://en.wikipedia.org/wiki/Linear_classifier)*, therefore it cannot learn non-linear relations between the input and the desired output.

So what is a multi-layer neural network and how to avoid the linear classifier trap?

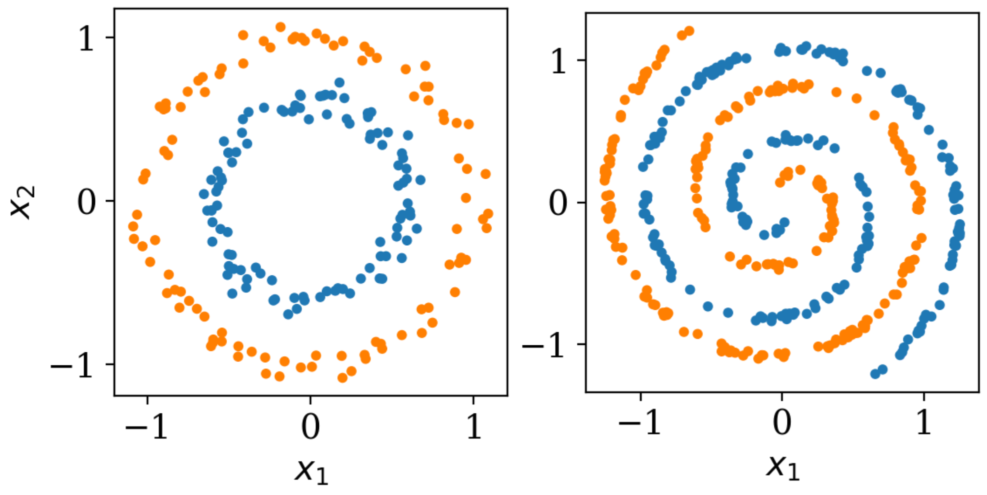
Multilayer neural networks

Multilayer neural network can be formalized as:

$$\mathbf{y} = f_N\left(\mathbf{W}_N \cdot \left(\ldots f_2\left(\mathbf{W}_2 \cdot f_1(\mathbf{W}_1 \cdot \mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2\right) \ldots\right) + \mathbf{b}_N\right), \qquad (1)$$

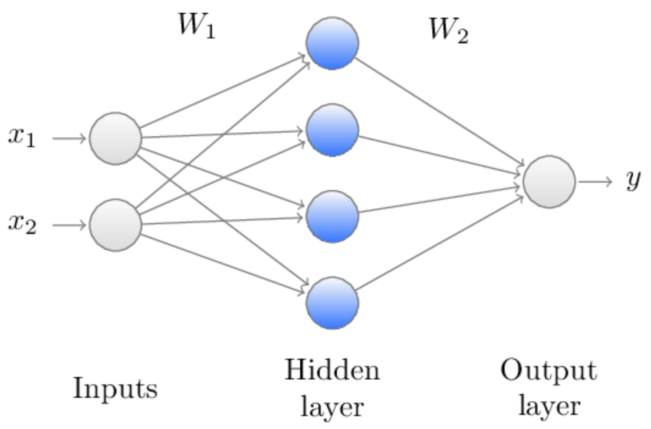
where $\mathbf{x}$ and $\mathbf{y}$ is an input and output vectors respectively, $\mathbf{W}_i, \mathbf{b}_i, i = 1..N$ are weight matrices and bias vectors, and $f_i, i = 1..N$ are activations functions in the ith layer. The nonlinear activation functions f_i are applied elementwise.

For the sake of illustration, today we will employ two-dimensional datasets as shown below.



Both datasets feature two classes depicted by orange and blue dots, respectively. The goal of the neural network training will be to learn how to prescribe a class to a new dot by knowing its coordinates (x_1, x_2). It turns out these simple tasks cannot be solved with a single-layer architecture, as it is only capable of drawing straight lines in the input space (x_1, x_2). Let us bring in an additional layer.

Two-layer network



Two-layer neural network.
Inputs x_1 and x_2 are multiplied by a weights matrix W_1, then activation function f_1 is applied element-wise. Finally, the data are transformed by W_2 followed by another activation f_2 (not depicted) to obtain the output y.

The figure above shows the single hidden layer network architecture, which we will interchangeably call a *two-layer network*:

$$y = f_2\left(\mathbf{W}_2 \cdot f_1(\mathbf{W}_1 \cdot \mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2\right),$$

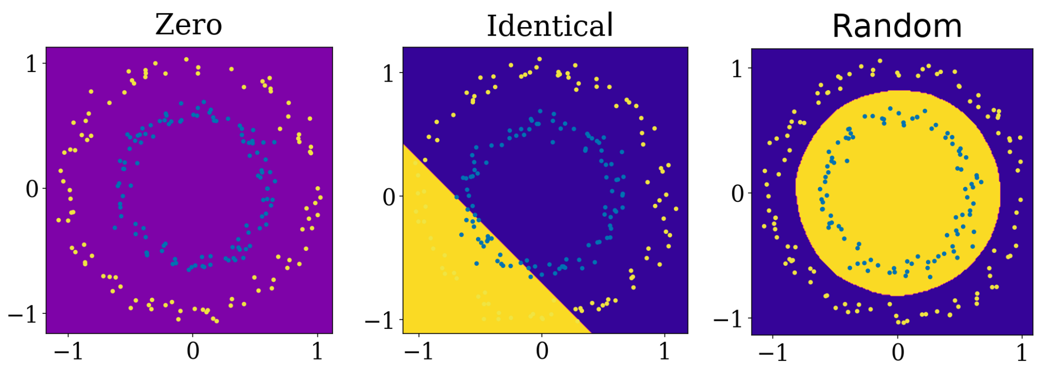
where $f_1(x) = \max(0, x)$ also known as ReLU (https://en.wikipedia.org/wiki/Rectifier_(neural_networks)) will be the first activation function and sigmoid $f_2(x) = \sigma(x) = [1 + e^{-x}]^{-1}$ will be the second activation function, which you already have seen in the previous article. The particularity of sigmoid is squashing its inputs between 0 and 1. This output tends to be useful in our case where we only have two classes: blue dots denoted by 0 and orange dots, by 1. As a loss function to minimize during neural network training, we will use *binary cross-entropy*:

$$L(y, \hat{y}) = -\frac{1}{m} \sum_{i=1}^{m} \left[y_i \ln(\hat{y}_i) + (1 - y_i) \ln(1 - \hat{y}_i) \right],$$

where y is the target output and $\hat{y}$ is neural network's prediction. This loss function was specifically designed (http://neuralnetworksanddeeplearning.com/chap3.html#the_cross-entropy_cost_function) to efficiently train neural networks. Another perk of cross-entropy is that in conjunction with sigmoid activation, the error gradient is simply a difference between predicted and desired outputs.

How To Initialize?

The first mistake may creep in when you completely ignore the importance of the neural network weight initialization (https://medium.com/usf-msds/deep-learning-best-practices-1-weight-initialization-14e5c0295b94). In essence, you want to create neurons capable of learning different features. Therefore, you should select neural network weights randomly.

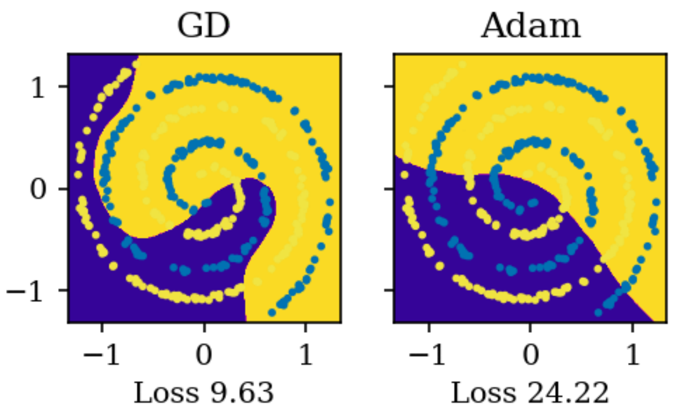


The circles problem.
Decision boundaries for neural networks trained using the gradient descent method and initialized with (1) zero weights, (2) identical weights, and (3) random weights.

Above we can see the decision boundaries (https://en.wikipedia.org/wiki/Decision_boundary) for three cases of initialization. First, when the weights are initialized as zeroes, the network cannot learn anything. In the second case, the weights are initialized as non-zero, but identical values. Even though the network becomes able to learn something, there is no distinction between individual neurons. Therefore, the entire network behaves like one big neuron. Finally, when the neural network is initialized with random weights, it becomes capable to learn.

How To Train?

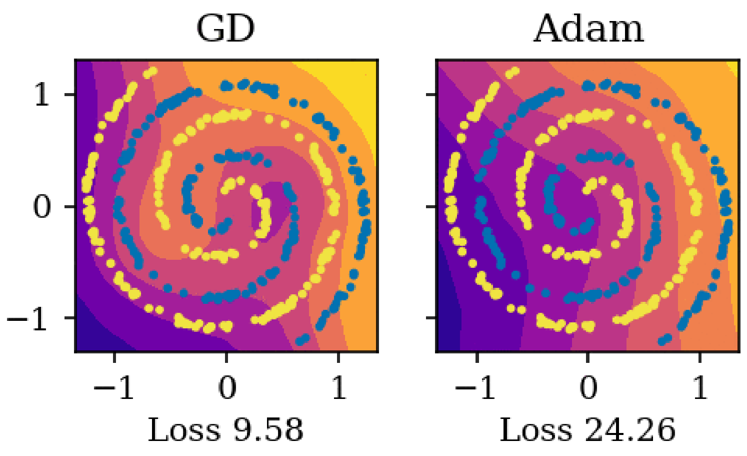
The universal approximation theorem (https://en.wikipedia.org/wiki/Universal_approximation_theorem) claims that under certain assumptions, a single hidden layer architecture should be sufficient to approximate any reasonable function. However, it indicates nothing about how to obtain or how difficult is to obtain those neural network weights[1]. As we have seen from weights initialization example, using simple gradient descent training our network was able to distinguish between the two circles. Now, how about a more difficult example: spirals, which *is hard due to its extreme nonlinearity* (https://www.gwern.net/docs/ai/1988-lang.pdf). Check out the animation below, on the left, to see how a naive gradient descent training can get easily stuck in a suboptimal setting.



The spirals problem.
Both neural networks have identical single hidden layer architecture with 512 neurons. The first one is trained with a naive gradient descent (GD), the second one is trained with Adam.

The network fails to properly distinguish between the two classes. One may suggest that a more adequate neural network architecture is needed. That is partially true. However, the mentioned above theorem (https://en.wikipedia.org/wiki/Universal_approximation_theorem) hints that it might be not necessary.

It turns out a lot of research in neural networks and deep learning is exactly about how to obtain those set of weights, that is about neural network training algorithms. The disadvantage of the naive gradient descent is that the algorithm often gets stuck far from an acceptable solution. One of the ways to alleviate this issue is to introduce momentum. Perhaps, the most popular among the algorithms with momentum is Adam (https://arxiv.org/abs/1412.6980). As we can see from the diagram above, on the right, the Adam (https://www.youtube.com/watch?v=JXQT_vxqwIs) algorithm is able to efficiently build the appropriate decision boundary. I find it can be more illustrative if we visualize the neural network's output during its training.



The spirals problem.
The output of two trained models before thresholding. One may see that the one trained with gradient descent (GD) is practically stuck in a suboptimal condition, whereas Adam algorithm efficiently leads to the separation between two classes.

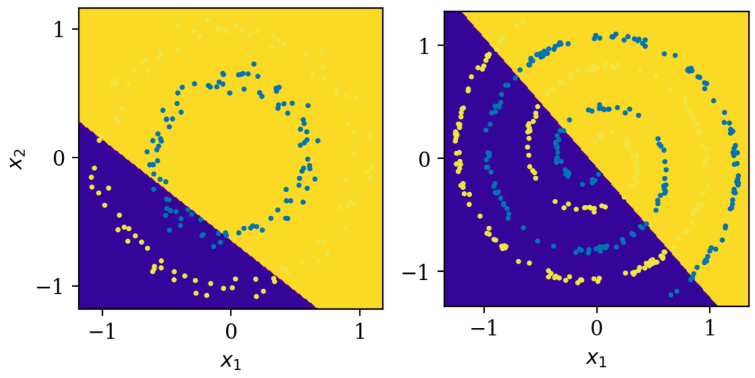
This way we see the faster convergence of Adam on a more detailed level. If you want to learn more about different algorithms from the gradient descent family, here (http://ruder.io/optimizing-gradient-descent/) there is a nice comparison.

Avoiding The Linear Classifier Pitfall

Now, what if by mistake we have provided linear activations. What happens? If activations f_i were linear, then the model in formula (2) would reduce to a single-layer neural network:

$$\mathbf{y} = c_N \cdot W_N \cdot \dots \cdot c_2 \cdot W_2 \cdot c_1 \cdot W_1 \cdot \mathbf{x} = W\mathbf{x},$$

where c_i denote constants. Thus, one obtains a linear classifier $W\mathbf{x}$. This can be seen from the resulting decision boundaries.

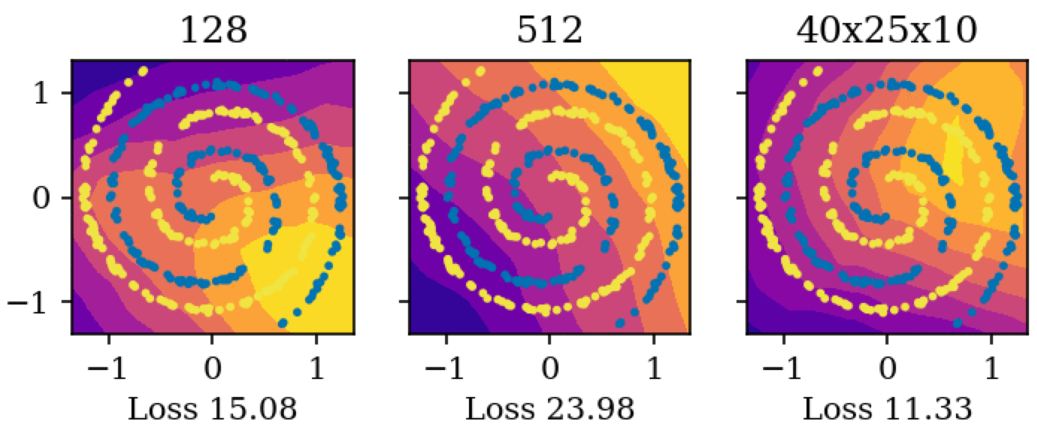


Two-layer network with linear activation $f_1(x) = x$.

We have used exactly the same architecture as above, except the activation $f_1(x) = x$. Now we see that the two-layer neural network acts as a simple single-layer one: it can only divide the input space using a straight line. To avoid this issue, all activation functions must remain nonlinear.

How Many Layers?

We have solved the spirals problem with a single hidden layer network. Why add more layers? To address this question, let us animate the decision boundaries during neural network training for three different architectures. All the networks have ReLU (https://en.wikipedia.org/wiki/Rectifier_(neural_networks)) activations in non-terminal layers and sigmoid in the final layer. As can be seen from the image below, the network with three hidden layers and with fewer neurons in each layer, learns faster. The reason can be understood intuitively. A neural network performs topological transformations (http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/) of the input space. Composing several simple transformations leads to a more complex one. Therefore, for such non-linearly separated classes as in the spirals problem one benefits from adding more layers.



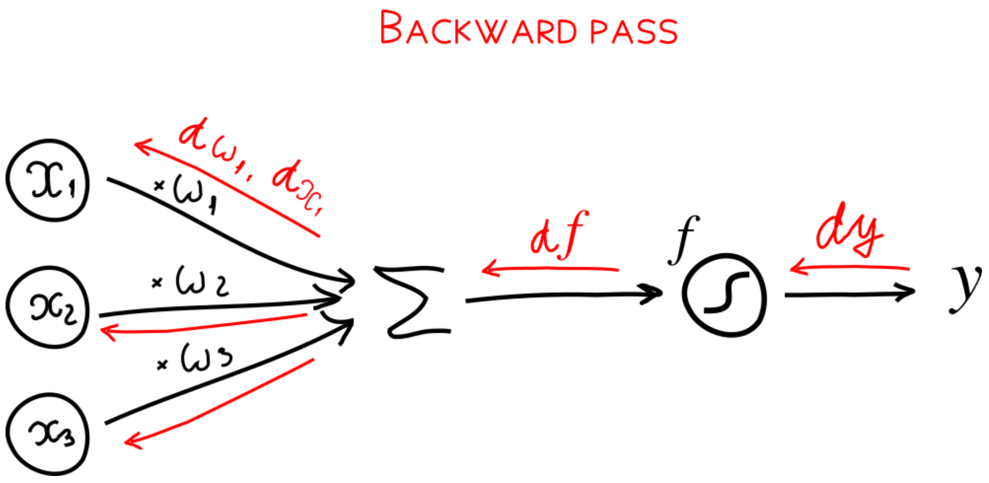
Neural network training convergence (Adam).
Architectures: (1) single hidden layer with 128 neurons (513 total parameters), (2) single hidden layer, 512 neurons (2049 parameters), (3) three hidden layers with 40, 25, and 10 neurons (1416 parameters). The fastest convergence is demonstrated by the last architecture.

From the figure we may conclude that more neurons are not always the best option. The most contrasting border indicating the highest confidence in the least number of steps is achieved with the three hidden layers neural network. This network has 1416 parameters counting both weights and biases. This number is about 30 percent less than the second architecture with a single hidden layer. Indeed, one may need to carefully pick (http://proceedings.mlr.press/v70/real17a/real17a.pdf) the (https://github.com/JasperSnoek/spearmint) architecture (https://aip.scitation.org/doi/10.1063/1.5039826) taking into consideration different factors such as dataset size, variance, input dimensionality, kind of the task, and others. With more layers, neural network can more efficiently represent complex relationships (http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/). On the other hand, if the number of layers is very large, one may need to apply different tricks in order to avoid such problems as fading gradients.

Given all the new information, how about experimenting with neural networks? In the rest of this post we will learn how to build multilayer networks in Haskell.

Implementing Backpropagation

In the previous article (/neural-networks/day1/), we have implemented *backpropagation* (shortly *backprop*) for a single-layer (no hidden layer) network. This method easily extends towards the case of multiple layers. Here is a nice video (https://www.youtube.com/watch?v=d14TUNcbn1k&index=4&list=PLC1qU-LWwrF64f4QKQT-Vg5Wr4qEE1Zxk) to provide you with more technical details. Below, we remind how backprop works for each individual neuron.



Backprop for a neuron. A neuron's computation $y = f(\sum_i w_i \cdot x_i)$ known as the *forward pass* is illustrated in black using a computational graph. The *backward pass* propagating the gradients in the opposite direction is depicted in red.

If you have ever found any other tutorial on backprop, chances are you will first implement the so-called *forward pass* and then, as a separate function, the *backward pass*. The forward pass calculates the neural network output, whereas the backward pass does the gradients bookkeeping. Last time we have provided an example of a forward pass in a single-layer architecture:

```
forward x w =
  let h = x LA.<> w
      y = sigmoid h
  in [h, y]
```

where `h = x LA.<> w` calculates $\mathbf{h} = \mathbf{x} \cdot \mathbf{W}^{\intercal 2}$ and `y = sigmoid h` is an elementwise activation $\mathbf{y} = \sigma(\mathbf{h})$. Here we have provided both the result of neural network computation `y` and an intermediate value `h`, which was subsequently used in the backward pass to compute the weights gradient `dW`:

```
    dE = loss' y y_target
    dY = sigmoid' h dE
    dW = linear' x dY
```

If there are multiple layers, the two passes (forward and backward) are typically computing all the intermediate results $\mathbf{h}_i = \mathbf{x}_{i-1}\mathbf{W}_i^\intercal + \mathbf{b}_i$ and $\mathbf{y}_i = f(\mathbf{x}_i)$. Then, these intermediate results are used in reverse (backward) order to compute weight gradients `dW`.

The strategy of storing intermediate results is typical for many tutorials in imperative languages (looking at you, Python). In fact, in functional languages you can do the same thing (https://themonadreader.files.wordpress.com/2013/03/issue214.pdf). The disadvantage of separate forward and backward implementations is that it becomes tedious to keep track of all the intermediate results and may require more mental effort to implement the backprop. I also feel that such backprop explanation is not very intuitive. The whole pattern of forward and backward passes may seem strangely familiar. Hey, that is actually recursion (https://en.wikipedia.org/wiki/Recursion_(computer_science))! Recursion is bread and butter of functional languages, so why not apply it to the backprop? Instead of defining two separate methods and manually managing all the intermediate results, we will define both in one place.

To begin our new multilayer networks project, we start with data structures. First, we define a data type that will represent a single layer.

```
-- Neural network layer: weights, biases, and activation
data Layer a = Layer (Matrix a) (Matrix a) Activation
```

where the first `Matrix a` stands for $\mathbf{W}^\intercal \in \mathbb{R}^{\mathrm{inp}\times\mathrm{out}}$[2], the second `Matrix a` means $\mathbf{b} \in \mathbb{R}^{1\times\mathrm{out}}$, and `Activation` is an algebraic data type, which in practice denotes known activation functions in a symbolic fashion:

```
-- Activation function:
-- * Rectified linear unit (ReLU)
-- * Sigmoid
-- * Hyperbolic tangent
-- * Identity (no activation)
data Activation = Relu | Sigmoid | Tanh | Id
```

Then, the neural network from Equation (2) can be serialized as a list of `Layer` s[3]. This list will bear the type of

```
[Layer Double]
```

indicating that each element in the list has to be a `Layer` operating on real numbers (`Double`).

Now, we can easily update any `Layer` in a single gradient descent step

```
f :: Layer Double
  -> Gradients Double
  -> Layer Double
f (Layer w b act) (Gradients dW dB) =
  Layer (w - lr `scale` dW) (b - lr `scale` dB) act
```

The signature `f :: Layer Double -> Gradients Double -> Layer Double` means nothing else but: *f is a function that can take an old layer and gradients and produce a new layer*. Here, `lr` is the learning rate and `scale` is a multiplication by a constant written in a fancy infix way[4]. We also have to define the `Gradients` data type:

```
data Gradients a = Gradients (Matrix a) (Matrix a)
```

To update all layers in functional programming we would need an analog to the `for` loop construct known from imperative programming languages. Great news: there are different `for` -loops for different needs: `map` (https://www.haskell.org/hoogle/?hoogle=map), `zip` (https://www.haskell.org/hoogle/?hoogle=zip), `zipWith` (https://www.haskell.org/hoogle/?hoogle=zipWith), `foldl` (https://www.haskell.org/hoogle/?hoogle=foldl), `scanr` (https://www.haskell.org/hoogle/?hoogle=scanr), etc. And `zipWith` (https://www.haskell.org/hoogle/?hoogle=zipWith) is exactly what we need. Its *polymorphic* (https://en.wikipedia.org/wiki/Polymorphism_(computer_science)) type

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

in our context can be interpreted as

```
zipWith
  :: (Layer a -> Gradients a -> Layer a)
     -> [Layer a]
     -> [Gradients a]
     -> [Layer a]
```

where `(Layer a -> Gradients a -> Layer a)` denotes a function which takes a `Layer` and `Gradients` and produces a new `Layer` . This is the signature of `f` defined above with the only precision that the type `a` is `Double` . Therefore, `zipWith` can be interpreted as: *take a function `f` from above, take a list of layers and a list of gradients, then apply `f` to each `Layer` in the first list and each `Gradient` in the second list. Obtain a new list of `Layer` s*. The new list of layers `[Layer]` is our modified neural network after a single gradient descent step. Now, the complete gradient descent implementation is

```
optimize lr iterN net0 dataSet = last $ take iterN (iterate step net0)
  where
    step net = zipWith f net dW
      where
        (_, dW) = pass net dataSet

    f :: Layer Double
      -> Gradients Double
      -> Layer Double
    f (Layer w b act) (Gradients dW dB) =
      Layer (w - lr `scale` dW) (b - lr `scale` dB) act
```

We now see how well previously defined `f` fits in. The resulting function `optimize` should look familiar from the previous post (/neural-networks/day1/). Back then, it was called `descend` . Now, we generalize it to operate on a list of layers, instead of a single layer weights. Finally, we provide the `dataSet` as an argument. This last interface change is a matter of convenience.

Now, as promised, we implement the backprop algorithm in a single `pass` .

```
80    pass net (x, tgt) = (pred, grads)
81      where
82        (_, pred, grads) = _pass x net
83
84        _pass inp [] = (loss', pred, [])
85          where
86            pred = sigmoid inp
87            -- Gradient of cross-entropy loss
88            -- after sigmoid activation
89            loss' = pred - tgt
90
91        _pass inp (Layer w b sact:layers) = (dX, pred, Gradients dW dB:t)
92          where
93            lin = (inp LA.<> w) + b
94            y = getActivation sact lin
95
96            (dZ, pred, t) = _pass y layers
97
98            dY = getActivation' sact lin dZ
99            dW = linearW' inp dY
100           dB = bias' dY
101           dX = linearX' w dY
```

Let us break down what the code above does. First, the inputs are an initial neural network `net` and training data with corresponding targets `(x, tgt)`. The function will provide inference result and gradients `(pred, grads)` as the computation result. Line 82 launches the `_pass` function which accepts two arguments: initial input `x` and neural network `net`.

```
91        _pass inp (Layer w b sact:layers) = (dX, pred, Gradients dW dB:t)
92          where
93            lin = (inp LA.<> w) + b
94            y = getActivation sact lin
95
96            (dZ, pred, t) = _pass y layers
97
98            dY = getActivation' sact lin dZ
99            dW = linearW' inp dY
100           dB = bias' dY
101           dX = linearX' w dY
```

This piece of code above performs computations with respect to the current layer. First, we perform pattern matching selecting current layer weights and biases `w b` and activation `sact` and also we get the tail of the layers list `layers` on line 91. Lines 93-94 perform the forward pass, i.e. the $f(\mathbf{Wx} + \mathbf{b})$ operation. Line 96 (highlighted) launches a recursive call to `_pass` providing the arguments of current layer computation `y` and the rest of the tail of the network `layers`. As a result of the recursive call we obtain gradient coming back from the layer `dZ`, prediction result `pred`, and partially computed weights gradients list `t`. Lines 98-101 compute new gradients, which `_pass` returns to its previous caller.

Now, what happens when we finish our forward propagation? Lines 84-89 below contain the stop condition for the recursive call. The empty list `[]` on line 84 signifies that we have reached the end of the layers list. Now, we compute the network prediction: We apply sigmoid to squash the outputs between 0 and 1 on line 86. This, together with a binary cross-entropy loss function, will allow for a very simple gradient (https://deepnotes.io/softmax-crossentropy) in the final layer: `loss' = pred - tgt` on line 89 (highlighted).

```
84      _pass inp [] = (loss', pred, [])
85        where
86          pred = sigmoid inp
87          -- We calculate the gradient of cross-entropy loss
88          -- after sigmoid activation.
89          loss' = pred - tgt
```

The function `_pass` provides three results: the loss gradient `loss'`, the predicted result `pred`, and an empty list `[]` to accumulate the weight gradients in the backward pass (lines 96-101). Note that since `sigmoid` activation has been already calculated on line 86, the final layer must have no activation, i.e. we will specify the identity transformation `Id` of the final layer, when defining the network architecture.

Now, the question: how can we calculate the forward pass separately, i.e. without the need to compute all the gradients? It turns out that Haskell has *lazy* evaluation: it computes only what we specifically ask for. In the `forward` implementation below, gradients are never asked, and therefore never computed.

```
forward net dta = fst $ pass net (dta, undefined)
```

Normally, `undefined` will return an error when encountered. However, since we never calculate the gradients, the undefined target is never evaluated.

Putting It All Together

Now, we have all the necessary components to create a multilayer neural network. First, we generate our data. Then, we define the architecture and initialize the actual neural networks. Finally, we run our experiments.

Train And Validation Datasets

We generate a random circles dataset for our experiments. We will provide the coordinate pairs (x_1, x_2) as well as target class labels y.

```
makeCircles
  :: Int -> Double -> Double -> IO (Matrix Double, Matrix Double)
makeCircles m factor noise = do

  -- Code omitted

  return (x, y)
```

In general, it is a good idea to have at least two separate machine learning datasets: one used for training and another one for model validation. Therefore, we will create two of those with randomly generated samples.

```
  trainSet <- makeCircles 200 0.6 0.1
  testSet <- makeCircles 100 0.6 0.1
```

We create the `trainSet` with 200 and the `testSet` with 100 data samples. `0.6` and `0.1` are model parameters, for the full implementation feel free to check Main.hs (http://github.com/penkovsky/10-days-of-grad/tree/master/day2). In a similar way we will create training and validation datasets for the spirals problem.

Defining The Architecture

Below we define an architecture for a single hidden layer network.

```
  -- Initializing a single hidden layer network
  let (nIn, hidden, nOut) = (2, 512, 1)

  -- Generate weights/biases W1 and W2
  (w1_rand, b1_rand) <- genWeights (nIn, hidden)
  (w2_rand, b2_rand) <- genWeights (hidden, nOut)

  let net = [ Layer w1_rand b1_rand Relu
            , Layer w2_rand b2_rand Id ]
```

Note that the activation of the final layer is calculated separately, therefore we leave no activation `Id`. `w*_rand` and `b*_rand` are randomly generated weights and biases. For convenience we define a `genNetwork` function which will combine the operations from above. This is how it is used:

```
net <- genNetwork [2, 512, 1] [Relu, Id]
```

Initializing Weights

We remember that for the greatest benefit of a neural network, the weights should be initialized randomly. Here we employ a common initialization strategy from this paper (https://arxiv.org/pdf/1502.01852.pdf).

```
-- Generate new random weights and biases
genWeights :: (Int, Int) -> IO (Matrix Double, Matrix Double)
genWeights (nin, nout) = do
  w <- _genWeights (nin, nout)
  b <- _genWeights (1, nout)
  return (w, b)
    where
      _genWeights (nin, nout) = do
          let k = sqrt (1.0 / fromIntegral nin)
          w <- randn nin nout
          return (k `scale` w)
```

This strategy becomes more useful as soon as your neural network becomes "deeper", i.e. gets more and more layers.

Adam

Previously we have seen that naive gradient descent may easily get stuck in a suboptimal setting. Let us implement Adam optimization strategy, explained in this video (https://www.youtube.com/watch?v=JXQT_vxqwIs).

```
adam p dataSet iterN w0 = w
  where

    -- Some code omitted

        (_, dW) = pass dataSet w loss'

        sN = zipWith f2 s dW
        vN = zipWith f3 v dW
        wN = zipWith3 f w vN sN

        f :: Layer Double
          -> (Matrix Double, Matrix Double)
          -> (Matrix Double, Matrix Double)
          -> Layer Double
        f (w_, b_, sf) (vW, vB) (sW, sB) =
          ( w_ - lr `scale` vW / ((sqrt sW) `addC` epsilon)
          , b_ - lr `scale` vB / ((sqrt sB) `addC` epsilon)
          , sf)
        addC m c = cmap (+ c) m

        f2 :: (Matrix Double, Matrix Double)
           -> Gradients Double
           -> (Matrix Double, Matrix Double)
        f2 (sW, sB) (dW, dB) =
          ( beta2 `scale` sW + (1 - beta2) `scale` (dW^2)
          , beta2 `scale` sB + (1 - beta2) `scale` (dB^2))

        f3 :: (Matrix Double, Matrix Double)
           -> Gradients Double
           -> (Matrix Double, Matrix Double)
        f3 (vW, vB) (dW, dB) =
          ( beta1 `scale` vW + (1 - beta1) `scale` dW
          , beta1 `scale` vB + (1 - beta1) `scale` dB)
```

The Adam implementation is a derived from `optimize` . The most prominent change is additional "zipping" with two new functions: `f2` embodies the RMSprop (https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf) strategy and `f3` performs momentum estimation.

Running the experiments

Now, we shall train our network:

```
  let epochs = 1000
      lr = 0.001  -- Learning rate
      net' = optimize lr epochs net trainSet
      netA = optimizeAdam adamParams epochs net trainSet
```

It is time to compile and run our networks. Please check the output below. The complete project (http://github.com/penkovsky/10-days-of-grad/tree/master/day2) is available on Github (http://github.com/penkovsky/10-days-of-grad/tree/master/day2). Feel free to play with neural network architecture and initialization strategies.

```
$ stack --resolver lts-10.6 --install-ghc ghc \
  --package hmatrix-0.18.2.0 \
  --package hmatrix-morpheus-0.1.1.2 \
  -- -O2 Main.hs
$ ./Main

Circles problem, 1 hidden layer of 128 neurons, 1000 epochs
---
Training accuracy (gradient descent) 76.5
Validation accuracy (gradient descent) 74.0

Training accuracy (Adam) 100.0
Validation accuracy (Adam) 96.0


Spirals problem, Adam, 700 epochs
---
1 hidden layer, 128 neurons (513 parameters)
Training accuracy 77.5
Validation accuracy 72.0

1 hidden layer, 512 neurons (2049 parameters)
Training accuracy 97.2
Validation accuracy 97.0

3 hidden layers, 40, 25, and 10 neurons (1416 parameters)
Training accuracy 100.0
Validation accuracy 100.0
```

The first experiment shows the advantage of Adam over a naive gradient descent training algorithm on a simple circles dataset. The second experiment compares three architectures solving the spirals challenge. I had to reduce the number of training epochs to 700 to show which architecture will converge first.

Summary

When designing and training neural networks one should keep in mind several simple heuristics.

- Initialize weights randomly.
- Keep activation functions nonlinear. Remember, a superposition of linear operators is a linear operator.
- Use more layers and more neurons if the application appears to be complex. Reduce the number of layers in the opposite case.
- Apply training strategies with momentum instead of naive gradient descent.

The universal approximation theorem claims that a single hidden layer neural network architecture should be sufficient to approximate any reasonable map. However, how to obtain the weights is a much harder question. For instance, one of the first networks (https://www.gwern.net/docs/ai/1988-lang.pdf) solving the tough spirals challenge with backpropagation had not one, but three hidden layers. Indeed, we have seen that due to the highly nonlinear relationships in the dataset, a three-layer network with fewer parameters converges faster than the one with a single hidden layer.

In the next posts we will take a look on automatic differentiation (/neural-networks/day3/) and regularization techniques. We will also discuss how Haskell compiler can help ensure that our neural network is correct. And we will ramp it up to convolutional networks (/neural-networks/day5/) allowing us to solve some interesting challenges. Stay tuned.

I would like to dedicate this article to Maxence Ernoult for his early feedback and constructive criticism. I would also like to thank Elena Prokopets for proofreading the article.

Further Reading

Haskell

- A great start to learning Haskell (http://learnyouahaskell.com/)
- Try Haskell tutorial (https://tryhaskell.org/)
- Another example NN in Haskell (https://github.com/saschagrunert/nn)
- Yet one more (https://github.com/quickdudley/hfnn)

Neural Networks And Backpropagation

- A brief tutorial on backprop (http://cs231n.stanford.edu/slides/2018/cs231n_2018_ds02.pdf)
- More intuition on backprop (http://cs231n.github.io/optimization-2/) and slides (http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture4.pdf)
- Derivation of gradients (http://arunmallya.github.io/writeups/nn/backprop.html)
- Another tutorial on derivatives (http://cs231n.stanford.edu/handouts/derivatives.pdf)
- Notes on softmax and crossentropy (https://deepnotes.io/softmax-crossentropy)
- A more or less complete introduction to deep learning (http://neuralnetworksanddeeplearning.com/)
- More about the spirals challenge (http://web.archive.org/web/20160706105446/http://benmargolis.com/compsci/ai/two_spirals_problem

1. Such a cheeky theorem! ^
2. The attentive reader may have noticed that in the code we apply $\mathbf{x}\mathbf{W}^\intercal$ rather than $\mathbf{W}\mathbf{x}$, which is a matter of convenience. In many libraries you will find this transposition is done in order to keep data samples index in 0-th dimension (e.g. matrix rows). Also we have introduced biases $\mathbf{b}$ as in $\mathbf{W}\mathbf{x} + \mathbf{b}$ or $\mathbf{x}\mathbf{W}^\intercal + \mathbf{b}$, which were last time omitted for brevity. ^
3. For now, there is nothing that prevents us from mismatching matrix dimensions, however, we will revisit this issue in the future. ^
4. To put simply, `` a `f` b `` is the same as `f a b`, or $f(a, b)$. ^

Deep Learning (http://penkovsky.com/tags/deep-learning/) Haskell (http://penkovsky.com/tags/haskell/)

Next: Day 3: Haskell Guide To Neural Networks (http://penkovsky.com/neural-networks/day3/)

Related

- Day 1: Learning Neural Networks The Hard Way (/neural-networks/day1/)
- Towards Binarized Neural Networks Hardware (/talk/icee2018/)
- Delay Differential Equations (/project/dde/)
- HMEP (/project/hmep/)