Handwritten digits from the MNIST dataset (http://yann.lecun.com/exdb/mnist)

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

Day 4: The Importance Of Batch Normalization

Bogdan Penkovsky

Jun 29, 2019 · 14 min read · 10 Days Of Grad (http://penkovsky.com/categories/10-days-of-grad/)

Which purpose do neural networks serve for? Neural networks are learnable models. Their ultimate goal is to approach or even surpass human cognitive abilities. As Richard Sutton puts it, *'The biggest lesson that can be read from 70 years of AI research is that general methods that leverage computation are ultimately the most effective'*. In his essay (http://incompleteideas.net/IncIdeas/BitterLesson.html), Sutton argues that only models without encoded human-knowledge can outperform human-centeric approaches. Indeed, neural networks are general enough and they leverage computation. Then, it is not surprising how they can exhibit millions of learnable degrees of freedom.

The biggest challenge with neural networks is two-fold: (1) how to train those millions of parameters, and (2) how to interpret them. Batch normalization (shortly *batchnorm*) was introduced (https://arxiv.org/abs/1502.03167) as an attempt to make training more efficient. The method can dramatically reduce the number of training epochs. Moreover, batchnorm is perhaps the key ingredient that made possible training of certain architectures such as binarized neural networks (https://papers.nips.cc/paper/6573-binarized-neural-networks.pdf). Finally, batchnorm is one of the most recent neural network advances[1].

Previous posts

- Day 1: Learning Neural Networks The Hard Way (/neural-networks/day1/)
- Day 2: What Do Hidden Layers Do? (/neural-networks/day2/)
- Day 3: Haskell Guide To Neural Networks (/neural-networks/day3/)

The source code from this post is available on Github (https://github.com/masterdezign/10-days-of-grad/tree/master/day4).

Batch Normalization In Short

What Is A Batch?

Until now, we have looked on toy datasets. These were so small that they could completely fit into the memory. However, in the real world exist huge databases occupying hundreds of gigabytes of memory, such as Imagenet (http://www.image-net.org/) for example. Those often would not fit into the memory. In that case, it makes more sense to split a dataset into smaller *mini-batches*. During a forward/backward pass, only one batch is typically processed.

As the name suggests, batchnorm transformation is acting on individual batches of data. The outputs of linear layers may cause activation function saturation/'dead neurons'. For instance, in case of ReLU (rectified linear unit) $f(x) = \max(0, x)$ activation, all negative values will result in zero activations. Therefore, it is a good idea to normalize those values by subtracting the batch mean μ. Similarly, division by standard deviation $\sqrt{\mathrm{var}}$ scales the amplitudes, which is especially beneficial for sigmoid (https://en.wikipedia.org/wiki/Sigmoid_function)-like activations.

Training And Batchnorm

The batch normalization procedure differs between the training and inference phases. During the training, for each layer where we want to apply batchnorm we first compute the mini-batch mean:

$$\mu = \langle \mathbf{X} \rangle = \frac{1}{m} \sum_{i=1}^{m} \mathbf{X}_i, \tag{1}$$

where $\mathbf{X}_i$ is the ith feature-vector coming from the previous layer; $i = 1 \ldots m$, where $m > 1$ is the batch size. We also obtain the mini-batch variance:

$$\mathrm{var} = \frac{1}{m} \sum_{i=1}^{m} (\mathbf{X}_i - \mu)^2. \tag{2}$$

Now, the batchnorm's heart, normalization itself:

$$\hat{\mathbf{X}}_i = \frac{\mathbf{X}_i - \mu}{\sqrt{\mathrm{var} + \epsilon}}, \tag{3}$$

where a small constant ϵ is added for numerical stability. What if normalization of the given layer was harmful? The algorithm provides two learnable parameters that in the worst case scenario can undo the effect of batch normalization: scaling parameter γ and shift β. After (optionally) applying those, we obtain the output of the batchnorm layer:

$$\mathbf{Y}_i = \gamma * \hat{\mathbf{X}}_i + \beta. \tag{4}$$

Please note that both mean μ and variance var are vectors of as many elements as neurons in a given hidden layer. Operator $*$ denotes element-wise multiplication.

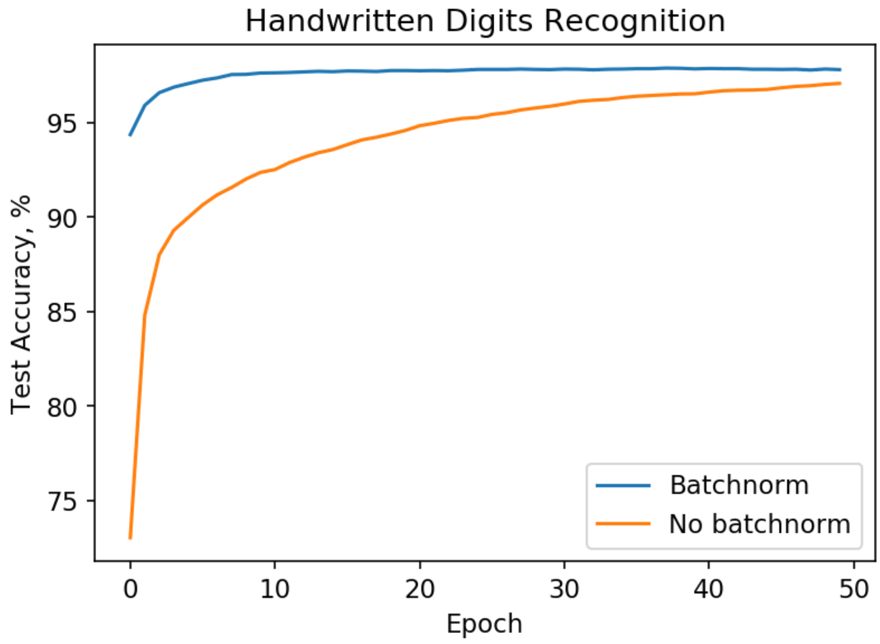
Inference

In the inference phase it is perfectly normal to have one data sample at a time. So how to calculate batch mean if the whole batch is a single sample? To properly handle this, during training we estimate mean and variance ($E[\mathbf{X}]$ and $\mathrm{Var}[\mathbf{X}]$) over all training set. Those vectors will replace μ and var during inference, avoiding thus the problem of normalizing a singleton batch.

How Efficient Is Batchnorm

So far we have played with tasks that provided low-dimensional input features. Now, we are going to test neural networks on a bit more interesting challenge. We will apply our skills to automatically recognize human-written digits on a famous MNIST dataset (http://yann.lecun.com/exdb/mnist/). This challenge came from the need to have zip-code machine reading (http://yann.lecun.com/exdb/publis/pdf/matan-92.pdf) for more efficient postal services.

We construct two neural networks, each having two fully-connected hidden layers (300 and 50 neurons). Both networks receive $28 \times 28 = 784$ inputs, the number of image pixels, and give back 10 outputs, the number of recognized classes (digits). As in-between layer activations we apply ReLU $f(x) = \max(0, x)$. To obtain the classification probabilities vector in the result, we use softmax (https://en.wikipedia.org/wiki/Softmax_function) activation $\sigma(\mathbf{x})_i = \frac{\exp x_i}{\sum_{j=1} \exp x_j}$. One of the networks in addition performs batch normalization before ReLUs. Then, we train those using stochastic gradient descent[2] with learning rate $\alpha = 0.01$[3] and batch size $m = 100$.



Neural network training on MNIST data.
Training with batchnorm (blue) leads to high accuracies faster than without batchnorm (orange).

From the figure above we see that the neural network with batch normalization reaches about 98% accuracy in ten epochs, whereas the other one is struggling to reach comparable performance in fifty epochs! Similar results can be obtained for other architectures.

It is worth mentioning that we still may lack understanding how exactly does batchnorm help. In its original paper (https://arxiv.org/abs/1502.03167), it was hypothesized that batchnorm is reducing internal covariate shift. Recently, it was shown (https://arxiv.org/abs/1805.11604) that that is not necessary true. The best up-to-date explanation (https://arxiv.org/abs/1805.11604) is that batchnorm makes optimization landscape smooth, thus making gradient descent training more efficient. This, in its turn, allows using higher learning rates (https://arxiv.org/abs/1502.03167) than without batchnorm!

Implementing batchnorm

We will base our effort on the code previously introduced on Day 2[4]. First, we will redefined the `Layer` data structure making it more granular:

```
data Layer a = -- Linear layer with weights and biases
               Linear (Matrix a) (Vector a)
               -- Same as Linear, but without biases
               | Linear' (Matrix a)
               -- Batchnorm with running mean, variance, and two
               -- learnable affine parameters
               | Batchnorm1d (Vector a) (Vector a) (Vector a) (Vector a)
               -- Usually non-linear element-wise activation
               | Activation FActivation
```

Amazing! Now we can distinguish between several kinds of layers: affine (linear), activation, and batchnorm. Since batchnorm already compensates for a bias, we do not actually need biases in the subsequent linear layers. That is why we define a `Linear'` layer without biases. We also extend `Gradients` to accommodate our new layers structure:

```
data Gradients a = -- Weight and bias gradients
                   LinearGradients (Matrix a) (Vector a)
                   -- Weight gradients
                   | Linear'Gradients (Matrix a)
                   -- Batchnorm parameters and gradients
                   | BN1 (Vector a) (Vector a) (Vector a) (Vector a)
                   -- No learnable parameters
                   | NoGrad
```

Next, we want to extend the neural network propagation function `_pass`, depending on the layer. That is easy with *pattern matching* (http://learnyouahaskell.com/syntax-in-functions). Here is how we match the `Batchnorm1d` layer and its parameters:

```
    _pass inp (Batchnorm1d mu variance gamma beta:layers)
        = (dX, pred, BN1 batchMu batchVariance dGamma dBeta:t)
      where
```

As previously, the `_pass` function receives an input `inp` and layer parameters. The second argument is the pattern we are matching against, making our algorithm specific in this case for `(Batchnorm1D ...)`. We will also specify `_pass` for other kinds of `Layer`. Thus, we have obtained a *polymorphic* `_pass` function with respect to the layers. Finally, the equation results in a tuple of three: gradients to back propagate `dX`, predictions `pred`, and prepended list `t` with values `BN1` computed in this layer (batch mean `batchMu`, variance `batchVariance`, and learnable parameters gradients).

The forward pass as illustrated in this post (https://kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalization-layer.html):

```
        -- Forward
        eps = 1e-12
        b = br (rows inp)  -- Broadcast (replicate) rows from 1 to batch size
        m = recip $ (fromIntegral $ rows inp)

        -- Step 1: mean from Equation (1)
        batchMu :: Vector Float
        batchMu = compute $ m `_scale` (_sumRows inp)

        -- Step 2: mean subtraction
        xmu :: Matrix Float
        xmu = compute $ inp .- b batchMu

        -- Step 3
        sq = compute $ xmu .^ 2

        -- Step 4: variance, Equation (2)
        batchVariance :: Vector Float
        batchVariance = compute $ m `_scale` (_sumRows sq)

        -- Step 5
        sqrtvar = sqrtA $ batchVariance `addC` eps

        -- Step 6
        ivar = compute $ A.map recip sqrtvar

        -- Step 7: normalize, Equation (3)
        xhat = xmu .* b ivar

        -- Step 8: rescale
        gammax = b gamma .* xhat

        -- Step 9: translate, Equation (4)
        out0 :: Matrix Float
        out0 = compute $ gammax .+ b beta
```

As discussed on Day 2 (/neural-networks/day2/), there is a recurrent call obtaining gradients from the next layer, neural network prediction `pred` and computed values tail `t`:

```
        (dZ, pred, t) = _pass out layers
```

I prefer to keep the backward pass without any simplifications. That makes clear which step corresponds to which:

```
        -- Backward

        -- Step 9
        dBeta = compute $ _sumRows dZ

        -- Step 8
        dGamma = compute $ _sumRows (compute $ dZ .* xhat)
        dxhat :: Matrix Float
        dxhat = compute $ dZ .* b gamma

        -- Step 7
        divar = _sumRows $ compute $ dxhat .* xmu
        dxmu1 = dxhat .* b ivar

        -- Step 6
        dsqrtvar = (A.map (negate. recip) (sqrtvar .^ 2)) .* divar

        -- Step 5
        dvar = 0.5 `_scale` ivar .* dsqrtvar

        -- Step 4
        dsq = compute $ m `_scale` dvar

        -- Step 3
        dxmu2 = 2 `_scale` xmu .* b dsq

        -- Step 2
        dx1 = compute $ dxmu1 .+ dxmu2
        dmu = A.map negate $ _sumRows dx1

        -- Step 1
        dx2 = b $ compute (m `_scale` dmu)

        dX = compute $ dx1 .+ dx2
```

Note that often we need to perform operations like mean subtraction $\mathbf{X} - \mu$, where in practice we have a matrix $\mathbf{X}$ and a vector μ. How do you subtract a vector from a matrix? Right, *you don't*. You can subtract only two matrices. Libraries like `Numpy` may have a *broadcasting* (https://docs.scipy.org/doc/numpy/user/theory.broadcasting.html) magic that would implicitly convert a vector to a matrix[5]. This broadcasting might be useful, but might also obscure different kinds of bugs. We instead perform explicit vector to matrix transformations. For our convenience, we have a shortcut `b = br (rows inp)` that will expand a vector to the same number of rows as in `inp`. Where function `br` ('broadcast') is:

```
br rows' v = expandWithin Dim2 rows' const v
```

Here is an example how `br` works. First, we start an interactive Haskell session and load `NeuralNetwork.hs` module:

```
$ stack exec ghci
GHCi, version 8.2.2: http://www.haskell.org/ghc/  :? for help
Prelude> :l src/NeuralNetwork.hs
```

Then, we test `br` function on a vector `[1, 2, 3, 4]`:

```
*NeuralNetwork> let a = A.fromList Par [1,2,3,4] :: Vector Float
*NeuralNetwork> a
Array U Par (Sz1 4)
  [ 1.0, 2.0, 3.0, 4.0 ]

*NeuralNetwork> let b = br 3 a
*NeuralNetwork> b
Array D Seq (Sz (3 :. 4))
  [ [ 1.0, 2.0, 3.0, 4.0 ]
  , [ 1.0, 2.0, 3.0, 4.0 ]
  , [ 1.0, 2.0, 3.0, 4.0 ]
  ]
```

As we can see, a new matrix with three identical rows has been obtained. Note that `a` has type `Array U Seq`, meaning that data are stored in an unboxed array. Whereas the result is of type `Array D Seq`, a so-called *delayed array*. This delayed array is not an actual array, but rather a *promise to compute*[6] an array in the future. In order to obtain an actual array residing in memory, use `compute`:

```
*NeuralNetwork> compute b :: Matrix Float
Array U Seq (Sz (3 :. 4))
  [ [ 1.0, 2.0, 3.0, 4.0 ]
  , [ 1.0, 2.0, 3.0, 4.0 ]
  , [ 1.0, 2.0, 3.0, 4.0 ]
  ]
```

You will find more information about manipulating arrays in `massiv` documentation (http://hackage.haskell.org/package/massiv/docs/Data-Massiv-Array.html). Similarly to `br`, there exist several more convenience functions, `rowsLike` and `colsLike`. Those are useful in conjunction with `_sumRows` and `_sumCols`:

```
-- | Sum values in each column and produce a delayed 1D Array
_sumRows :: Matrix Float -> Array D Ix1 Float
_sumRows = A.foldlWithin Dim2 (+) 0.0

-- | Sum values in each row and produce a delayed 1D Array
_sumCols :: Matrix Float -> Array D Ix1 Float
_sumCols = A.foldlWithin Dim1 (+) 0.0
```

Here is an example of `_sumCols` and `colsLike` when computing softmax (https://en.wikipedia.org/wiki/Softmax_function) activation $\sigma(\mathbf{x})_i = \frac{\exp x_i}{\sum_{j=1} \exp x_j}$:

```
softmax :: Matrix Float -> Matrix Float
softmax x =
  let x0 = compute $ expA x :: Matrix Float
      x1 = compute $ (_sumCols x0) :: Vector Float
      x2 = x1 `colsLike` x
  in (compute $ x0 ./ x2)
```

Note that softmax is different from element-wise activations. Instead, softmax acts as a fully-connected layer that receives a vector and outputs a vector. Finally, we define our neural network with two hidden linear layers and batch normalization as:

```
  let net = [ Linear' w1
            , Batchnorm1d (zeros h1) (ones h1) (ones h1) (zeros h1)
            , Activation Relu
            , Linear' w2
            , Batchnorm1d (zeros h2) (ones h2) (ones h2) (zeros h2)
            , Activation Relu
            , Linear' w3
            ]
```

The number of inputs is the total number of $28 \times 28 = 784$ image pixels and the number of outputs is the number of classes (ten digits). We randomly generate the initial weights `w1` , `w2` , and `w3` . And set initial batchnorm layer parameters as follows: means to zeroes, variances to ones, scaling parameters to ones, and translation parameters to zeroes:

```
  let [i, h1, h2, o] = [784, 300, 50, 10]
  (w1, b1) <- genWeights (i, h1)
  let ones n = A.replicate Par (Sz1 n) 1 :: Vector Float
      zeros n = A.replicate Par (Sz1 n) 0 :: Vector Float
  (w2, b2) <- genWeights (h1, h2)
  (w3, b3) <- genWeights (h2, o)
```

Remember that the number of batchnorm parameters equals to the number of neurons. It is a common practice to put batch normalization before activations, however this sequence is not strict: one can put batch normalization after activations too. For comparison, we also specify a neural network with two hidden layers without batch normalization.

```
  let net2 = [ Linear w1 b1
             , Activation Relu
             , Linear w2 b2
             , Activation Relu
             , Linear w3 b3
             ]
```

In both cases the output softmax activation is omitted as it is computed together with loss gradients in the final recursive call in `_pass` :

```
    _pass inp [] = (loss', pred, [])
      where
        pred = softmax inp
        loss' = compute $ pred .- tgt
```

Here, `[]` on the left-hand side signifies an empty list of input layers, and `[]` on the right-hand side is the empty tail of computed values in the beginning of the backward pass.

The complete project is available on Github (https://github.com/masterdezign/10-days-of-grad/tree/master/day4). I recommend playing with different neuron networks architectures and parameters. Have fun!

Batchnorm Pitfalls

There are several potential traps when using batchnorm. First, batchnorm is different during training (https://github.com/masterdezign/10-days-of-grad/blob/389ee62a89a31946bab226731432345bdeaf3288/day4/src/NeuralNetwork.hs#L305) and during inference (https://github.com/masterdezign/10-days-of-grad/blob/389ee62a89a31946bab226731432345bdeaf3288/day4/src/NeuralNetwork.hs#L355). That makes your implementation more complicated. Second, batchnorm may fail (https://www.alexirpan.com/2017/04/26/perils-batch-norm.html) when training data come from different datasets. To avoid the second pitfall, it is essential to ensure that every batch represents the whole dataset, i.e. it has data coming from the same distribution as the machine learning task you are trying to solve. Read more (https://www.alexirpan.com/2017/04/26/perils-batch-norm.html) about that.

Summary

Despite its pitfalls, batchnorm is an important concept and remains a popular method in the context of deep neural networks. Batchnorm's power is that it can substantially reduce the number of training epochs or even help achieving better neural network accuracy. After discussing this, we are prepared for such hot subjects as convolutional neural networks (/neural-networks/day5/) and reinforcement learning. Stay tuned!

Further Reading

AI And Neural Networks

- Richard Sutton. The Bitter Lesson (http://incompleteideas.net/IncIdeas/BitterLesson.html)
- Using neural nets to recognize handwritten digits (http://neuralnetworksanddeeplearning.com/chap1.html)
- Batch normalization paper (https://arxiv.org/abs/1502.03167)
- How Does Batch Normalization Help Optimization? (https://arxiv.org/abs/1805.11604)

- On The Perils of Batch Norm (https://www.alexirpan.com/2017/04/26/perils-batch-norm.html)

Haskell

- Why I think Haskell is the best general purpose language (http://www.philipzucker.com/why-i-as-of-june-22-2019-think-haskell-is-the-best-general-purpose-language-as-of-june-22-2019/)
- An opinionated beginner's guide to Haskell in mid 2019 (https://github.com/theindigamer/not-a-blog/blob/master/opinionated-haskell-guide-2019.md)
- Massiv Array Library (http://hackage.haskell.org/package/massiv)
- Alexey's presentation about massiv (https://www.youtube.com/watch?v=AAx2a0bUsxA)

1. Most of the currently used neural network techniques date back to 80s. They became widely popular only now because of faster hardware and large datasets availability. The distinctively new methods are few: long short-term memory (1997), generative adversarial networks (2014), batchnorm (2015). ^
2. The difference of stochastic gradient descent is that it approximates the true gradient by a gradient over a mini-batch. ^
3. We have previously called learning rate γ. In this post we call learning rate α since γ is a scaling parameter in the original batch normalization paper (https://arxiv.org/abs/1502.03167). ^
4. There are some technical differences, though. Namely, we use massiv (http://hackage.haskell.org/package/massiv) library supporting multidimensional arrays and parallelism. For the reference, here (https://github.com/masterdezign/10-days-of-grad/tree/master/massiv) are the first two days reimplemented with `massiv`. Those multidimensional arrays are crucial when dealing with convolutional neural networks for image processing. ^
5. In general, broadcasting is about making multidimensional arrays with different shapes compatible, not only vector to matrix translation. ^
6. These delayed arrays are useful for computing optimizations such as *fusion*. ^

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

Next: Day 5: Convolutional Neural Networks Tutorial (http://penkovsky.com/neural-networks/day5/)

Related

- Day 3: Haskell Guide To Neural Networks (/neural-networks/day3/)

- Day 2: What Do Hidden Layers Do? (/neural-networks/day2/)
- Day 1: Learning Neural Networks The Hard Way (/neural-networks/day1/)
- Energy-efficient AI (/project/edge-ai/)
- Stochastic Computing for Hardware Implementation of Binarized Neural Networks (/publication/stochastic-bnn/)