GRADIENTS, BATCH NORMALIZATION AND LAYER NORMALIZATION

Edit

In this post, we will take a look at common pit falls with optimization and solutions to some of these issues. The main topics that will be covered are:

- Gradients
- Exploding gradients
- Vanishing gradients
- LSTMs (pertaining to vanishing gradients)
- Normalization

And then we will see how to implement batch and layer normalization and apply them to our cells.

GRADIENTS

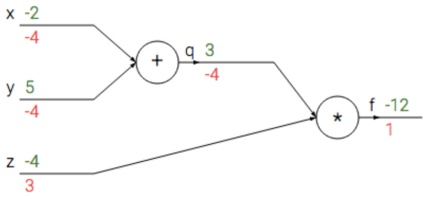
First, we will take a closer look at gradients and backpropagation during optimization. Our example will be a simple MLP but we will extend to an RNN later on.

I want to go over what a gradient means. Let's say we have a very simple MLP with 1 set of weights W_1 which is used to calcualte some y. We devise a very simple loss function J, and our gradient becomes dJ/dW_1 (d = partials). Sure we can take the derivative and apply chain rule and get a number, but what does this value even mean? The gradient can be thought of as several things. One is that the magnitude of the gradient represents the sensitivity or impact this weight has on determining y which determines our loss. This can be seen below:

```
# set some inputs
x = -2; y = 5; z = -4

# perform the forward pass
q = x + y # q becomes 3
f = q * z # f becomes -12

# perform the backward pass (backpropagation) in reverse order:
# first backprop through f = q * z
dfdz = q # df/dz = q, so gradient on z becomes 3
dfdq = z # df/dq = z, so gradient on q becomes -4
# now backprop through q = x + y
dfdx = 1.0 * dfdq # dq/dx = 1. And the multiplication here is the chain rule!
dfdy = 1.0 * dfdq # dq/dy = 1
```



CS231n

What the gradients (dfdx, dfdy, dfdz, dfdq, dfdz) tell us is the sensitivity of each variable on our result f. In an MLP, we will produce a result (logits) and compare it with our targets to determine the deviance in what we got and what we should have gotten. From this we can use backpropagation to determine how much adjusting needs to be made for each variable along the way, all the way to the beginning.

The gradient also holds another key piece of information. It repesents how much we need to change the weights in order to move towards our goal (minimizing the loss, maximizing some objective, etc.). With simple SGD, we get the gradient and we apply an update to the weights (W_i_new = W_i_old – alpha * gradient). If we follow the direction of the gradient, we will be maximizing the goal function. Our loss functions (NLL or cross entropy) are functions we wish to minimize, so we subtract the gradient. We use the learning parameter alpha to control how quickly we change. This is where all of the normalization techniques in this post will come in handy.

- SEARCH -

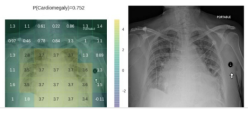
Search …

- FOLLOW ME ON TWITTER -

Tweets by @GokuMohandas

Goku Mohandas
@GokuMohandas

Your model may be performing really well by incorrectly focusing on confounding features (extraneous influencers in the data that aren't accounted for). Check out @johnrzech's post where x-ray stickers unintentionally influenced the classifications: medium.com/@jrzech/what-a…



Embed View on Twitter

- RECENT POSTS -

Update on Embeddings (Spring 2017)

Exploring Sparsity in Recurrent Neural Networks

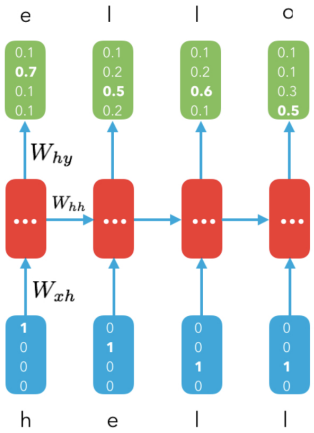
Question Answering from Unstructured Text by Retrieval and Comprehension

Overcoming Catastrophic Forgetting in Neural Networks

Opening the Black Box of Deep Neural Networks via Information

If we have an alpha that is 1 or larger, we will allow the gradient to directly impact our weights. In the beginning of training a neural net, our weight initializations are bound to be far off from the weights we actually need. This creates a large error and so, results in large gradients. If we choose to update our weights with these large gradients, we will be never reach the minimum point for our loss function. We will keep overshooting and bouncing back and forth. So, we use this alpha (small value) to control how much impact the gradient has. Eventually, the gradient will get smaller as well because of less error and we will reach our goal, but with such a small alpha, this can take a while. With techniques, such as batch normalization and layer normalization, we can afford to use large alpha because the gradients will be controlled due to controlled outputs from the neurons.

Now, even with a simple RNN structure, backpropagation can pose several issues. When we get our result, we need to backpropagate all the way back to the very first cell in order to complete our updates. The main principles to really understand are: if I multiply a number greater than 1 over and over, I will reach infinity (explosion) and vice versa, if I multiply a number less than 1 over and over, I will reach 0 (vanishing).



Forward Pass:

$$X_t$$

$$h_t = tanh(W_{hh}h_{t-1} + W_{xh}X_t + b_h)$$

$$y_t = W_{hy}h_t + b_y$$

$$P = \frac{e^y}{\sum e_y}$$

$$L = -h(P)$$

X: [NXC] W_{xh}: [CXH]
H: [NXH] W_{hh}: [HXH]
y: [NXC] W_{hy} : [HXC]

EXPLODING GRADIENTS

The first issue is that our gradients can be greater than 1. As we backpropagate the gradient through the network, we can end up with massive gradients. So far, the solution to exploding gradients is a very hacky but cheap solution; just clip the norm of the gradient at some threshold.

$$\nabla W \leftarrow \nabla W \frac{threshold}{max(|\nabla W|, threshold)}$$

VANISHING GRADIENTS

We could also experience the other issue where the gradient is less than 1 to start with and as we backpropagate, the effect of the gradient weakens and it will eventually be negligible. A common scenario where this occurs is when we have saturation at the tails of the sigmoidal function (0 or 1). This is problematic because now the derivative will always be near 0. During backpropagation, we will be multiplying this near zero derivative with our error repeatedly.

Let's look at the sigmoidal activation function. You can replicate this example for tanh too.

$$S(t) = \frac{1}{1 + e^{-z}}$$

$$S'(t) = S(t)(1 - S(t))$$

To solve this issue, we can use rectified linear units (ReLU) which don't suffer from this tail saturation as much.

$$f(x) = max(0, x)$$

The derivative is 1 if x > 0, so now error signal won't weaken as it backpropagates through the network. But we do have the problem in the negative region (x <0) where the derivative is zero. This can nullify our error signal so it's best to add a leaky factor (http://arxiv.org/abs/1502.01852) to the ReLU unit, where the negative region will have some small negative slope. This parameter can be fixed or be a randomized parameter and be fixed after training. There's also maxout (http://arxiv.org/abs/1302.4389) but this will have twice the amount of weights as a regular ReLU unit.

LSTMS (VANISHING GRADIENTS)

As for how LSTMs solve the vanishing gradient issue, they don't have to worry about the error signal weakening as with a regular basic RNN cell. It's a bit complicated but the basic idea is that they have a forget gate that determines how much previous memory is stored in the network

they have a forget gate that determines how much previous memory is stored in the network. This architecture allows the error signal to be transferred effectively to the previous time step. This is usually referred to as the constant error carousel (CEC).

NORMALIZATION

There are several types of normalization techniques but the idea behind all of them is the same, which is shifting our inputs to a zero mean and unit variance. We normalize the inputs before applying the non-linearity. We do this because we do not want the inputs to saturate the non-linearities at the extremes. (Checkout SNNs/SELU for some recent updates on this subject).

Techniques like batch norm (https://arxiv.org/abs/1502.03167) may help with the gradient issues as a side effect but the main object is to improve overall optimization. When we first initialize our weights, we are bound to have very large deviances from the true weights. These outliers need to be compensated for by the gradients and this further delays convergence during training. Batchnorm helps us here by normalizing the gradients (reducing influence from weight deviances) on a batched implementation and allows us to train faster (can even safely use larger learning rates now).

With batch norm, the main idea is to normalize at each layer for every minibatch. We initially may normalize our inputs, but as they travel through the layers, the inputs are operated on by weights and neurons and effectively change. As this progresses, the deviances get larger and larger and our backpropagation will need to account for these large deviances. This restricts us to using a small learning rate to prevent gradient explosion/vanishing. With **batch norm**, we will normalize the inputs (**activations** coming from the previous layer) going into each layer using the mean and variance of the activations for the **entire minibatch**. The normalization is a bit different during training and inference but it is beyond the scope of this post. (details in paper).

Batch normalization is very nice but it is based on minibatch size and so it's a bit difficult to use with recurrent architectures. With **layer normalization**, we instead compute the mean and variance using ALL of the summed inputs to the neurons in a layer for EVERY **single training case**. This removes the dependency on a minibatch size. Unlike batch normalization, the normalization operation for layer norm is same for training and inference. More details can be found on Hinton's paper **here**.

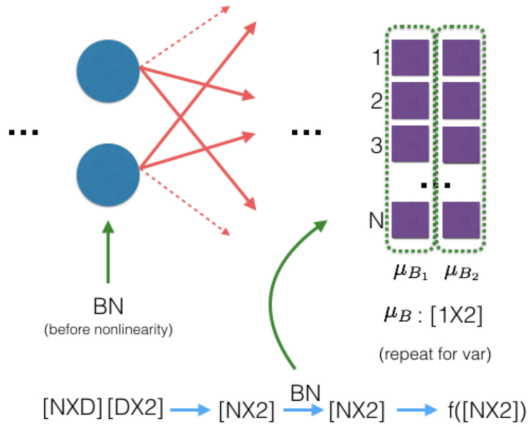
IMPLEMENTING BATCH NORMALIZATION

As stated above, the main goal of batch normalization is optimization. By normalizing the inputs to a layer to zero mean and unit variance, we can help our net learn faster by minimizing the effects from large errors (especially during initial training).

Batch norm is given by the operation below, where \epsilon is a small random noise (for stability). When we apply batch norm on a layer, we are restricting the inputs to follow a normal distribution, which ultimately will restrict the nets ability to learn. In order to fix this, we multiply by a scale parameter (\alpha) and add a shift parameter (\beta). Both of these parameters are trainable.

$$BN(x_i) = \alpha \odot (\frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}) + \beta$$

Note that both alpha and beta are applied element wise, so there will be a scale and shift for each neuron in the subsequent layer. With batchnorm, we compute mean and variance across an entire batch and we have a value for each neuron we are feeding our normalized inputs into.



So for a given layer, the mean during BN will be 1X. Each training data gets this mean subtracted from it and divided by sqrt(var + epsilon) and then shifted and scaled. To find the mean and var, we use all the examples in the training batch.

In order to accurately evaluate the effectiveness of batchnorm, we will use a simple MLP to classify MNIST digits. We will run a normal MLP and an MLP with batchnorm, both initialized with the same starting weights. Let's take a look at both the naive and TF implementations

the same starting weights. Let's take a look at both the naive and TF implementations.

First, the naive version:

```
# Naive BN layer
scale1 = tf.Variable(tf.ones([100]))
shift1 = tf.Variable(tf.zeros([100]))
W1_BN = tf.Variable(W1_init)
b1_BN = tf.Variable(tf.zeros([100]))
z1_BN = tf.matmul(X,W1_BN)+b1_BN
mean1, var1 = tf.nn.moments(z1_BN, [0])
BN1 = (z1_BN - mean1) / tf.sqrt(var1 + FLAGS.epsilon)
BN1 = scale1*BN1 + shift1
fc1_BN = tf.nn.relu(BN1)
```

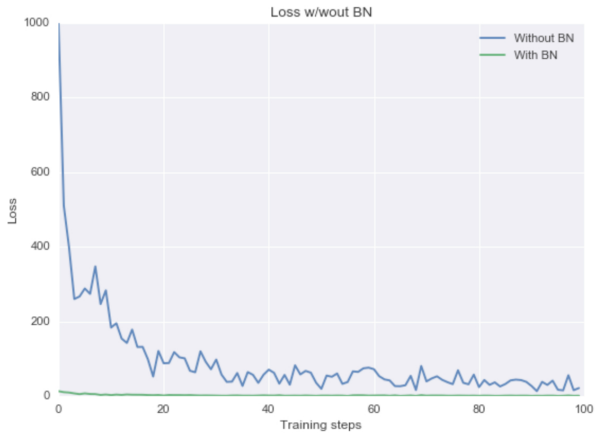
TF implementation:

```
# TF BN layer
scale2 = tf.Variable(tf.ones([100]))
shift2 = tf.Variable(tf.zeros([100]))
W2_BN = tf.Variable(W2_init)
b2_BN = tf.Variable(tf.zeros([100]))
z2_BN = tf.matmul(fc1_BN,W2_BN)+b2_BN
mean2, var2 = tf.nn.moments(z2_BN, [0])
BN2 = tf.nn.batch_normalization(z2_BN,mean2,var2,shift2,sca
fc2_BN = tf.nn.relu(BN2)
```

We first need to compute the mean and variance of the inputs coming into the layer. Then normalize them and scale/shift and then apply the activation function and pass to the next layer.

Let's compare the performance of the normal MLP and the MLP with batchnorm. We will focus of the massive impact on our cost with and without BN. Other interesting features to look at would be gradient norm, neuron inputs, etc.

CROSS ENTROPY LOSS



NUANCE:

Training is all fine and well, but what about testing. When doing BN on our test set, with the implementation from above, we will be using the mean and variance from our test set. Now think about what will happen if our test set is very small or even size 1. This will homogenize all the outputs we get since all inputs will be close to mean 0 and variance 1. The solution to this is to calculate the population mean and variance during testing and then use those values during testing.

Now there are couple ways we can try to calculate the population, even simple as taking the average of the training batch and using it for testing. This isn't the true population measure so we will calculate the unbiased mean and variance as they do in the original **paper**. But first, let's see the accuracy when we feed in test samples of size 1.

```
MLP Accuracy: 0.928
MLP w/ BN Accuracy: 0.969
Inference accuracy: 0.150
```

Not exactly state of the art anymore. So let's see how to calculate population mean and variance.

Population	Batch (sample)
$E(x) =$	$E_B[\mu_B]$
$Var(x) =$	$\frac{m}{m-1} E_B[\sigma_B^2]$

We will be updating the population mean and variance after each training batch and we will use them for inference. In fact we can simple replace the inference batchnorm process with a simple linear transformation:

$$BN(x_i) = \alpha \odot (\frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}) + \beta$$
$$= \frac{aX_i}{\sqrt{(\sigma_B^2 + \epsilon)}} - \frac{a\mu_B}{\sqrt{Var(x) + \epsilon}} + \beta$$
$$= \frac{aX_i}{\sqrt{(\sigma_B^2 + \epsilon)}} - \frac{aE(x)}{\sqrt{Var(x) + \epsilon}} + \beta$$
$$= \frac{a}{\sqrt{(\sigma_B^2 + \epsilon)}} X + (\beta - \frac{aE(x)}{\sqrt{Var(x) + \epsilon}})$$

Below is the tensorflow implementation for batchnorm with the exponential moving average to use during inference. Take a look **here** for more implementation specifications for batch_norm but the required parameters for us is the actual input that we wish to normalize and wether or not we are training. Note: TF batchnorm with inference is in **batch_norm2.py**

```
from tensorflow.contrib.layers import (
    batch_norm
)
...
with tf.variable_scope('BN_1') as BN_1:
    self.BN1 = tf.cond(self.is_training_ph,
        lambda: batch_norm(
            self.z1_BN, is_training=True, center=True,
            scale=True, activation_fn=tf.nn.relu,
            updates_collections=None, scope=BN_1),
        lambda: batch_norm(
            self.z1_BN, is_training=False, center=True,
            scale=True, activation_fn=tf.nn.relu,
            updates_collections=None, scope=BN_1, reuse=Tr
```

Here are the inference results with the population mean and variance:

```
MLP Accuracy: 0.921
MLP w/ BN Accuracy: 0.971
Inference accuracy: 0.950
```

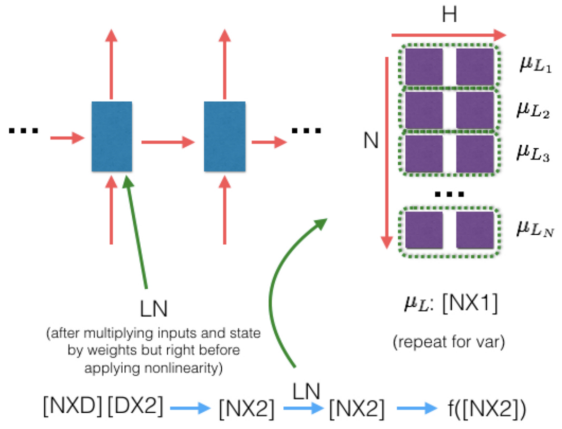
IMPLEMENTING LAYER NORMALIZATION

Layernorm is very similar to batch normalization in many ways as you can see with the equation below but it usually reserved for use with recurrent architectures.

$$LN = \alpha \odot \frac{x_i - u_L}{\sqrt{\sigma_L^2 + \epsilon}} + \beta$$

Layernorm acts on a per layer per sample basis, where the mean and variance are calculated for a specific layer for a specific training point. To understand the different between layernorm and batchnorm let's see how these mean and variances are computed for both with figures.

With layernorm it's a bit different from BN. We compute the mean and var for every single sample for each layer independently and then do the LN operations using those computed values.



First, we will make a function that will apply batch norm given an input tensor.

```
# LN funcition
def ln(inputs, epsilon = 1e-5, scope = None):

    """ Computer LN given an input tensor. We get in an in
    [N X D] and with LN we compute the mean and var for ea
    training point across all it's hidden dimensions rathe
    the training batch as we do in BN. This gives us a mea
    [N X 1].
    """
    mean, var = tf.nn.moments(inputs, [1], keep_dims=True)
```

```
    with tf.variable_scope(scope + 'LN'):
        scale = tf.get_variable('alpha',
                                shape=[inputs.get_shape()[
                                initializer=tf.constant_in
        shift = tf.get_variable('beta',
                                shape=[inputs.get_shape()[
                                initializer=tf.constant_in
    LN = scale * (inputs - mean) / tf.sqrt(var + epsilon)

    return LN
```

Now we can apply our LN function to a GRUCell class. Note that I am using tensorflow's **GRUCell class** but we can apply LN to all of their other RNN variants as well (LSTM, peephole LSTM, etc.)

```
class GRUCell(RNNCell):
  """Gated Recurrent Unit cell (cf. http://arxiv.org/abs/1

  def __init__(self, num_units, input_size=None, activatio
    if input_size is not None:
      logging.warn("%s: The input_size parameter is deprec
    self._num_units = num_units
    self._activation = activation

  @property
  def state_size(self):
    return self._num_units

  @property
  def output_size(self):
    return self._num_units

  def __call__(self, inputs, state, scope=None):
    """Gated recurrent unit (GRU) with nunits cells."""
    with vs.variable_scope(scope or type(self).__name__):
      with vs.variable_scope("Gates"):  # Reset gate and u
        # We start with bias of 1.0 to not reset and not u
        r, u = array_ops.split(1, 2, _linear([inputs, stat
                                              2 * self._num

        # Apply Layer Normalization to the two gates
        r = ln(r, scope = 'r/')
        u = ln(r, scope = 'u/')

        r, u = sigmoid(r), sigmoid(u)
      with vs.variable_scope("Candidate"):
        c = self._activation(_linear([inputs, r * state],
                                     self._num_units, True
      new_h = u * state + (1 - u) * c
    return new_h, new_h
```

SHAPES:

I received quite a few PMs about some confusing aspects of BN and LN, mostly centered around what is actually the input. Let's look at BN first. The input to a hidden layer will be [NXH]. Applying BN involves calculating the mean value for each H across all N samples. So we will have a mean of shape [1XH]. This "batch" mean will be used for BN, basically subtracting this batch mean from each sample.

Now for LN, let's imagine a simple RNN situation. Batch major inputs are of shape [N, M, H], where N is the batch size, M is the max number of time steps and H is the number of hidden units. Before feeing to an RNN, we can reshape to time-major which becomes [M, N, H]. Now we feed in one time step at a time into the RNN, so the shape of each time-step's input is [N,H]. Applying LN involves calculating the mean for sample across dimension [1], which means looking at all hidden states for each sample (for this particular time step). This gives us a mean of size [NX1]. We use this "layer" mean for each sample.

CODE:

Github Repo (Updating all repos, will be back up soon!)

SELU Code

Posted in: *Optimization/Architecture*
Tagged: *Tutorials*

← *Image Recognition* *Generative Adversarial Text to Image Synthesis* →

5 THOUGHTS ON "GRADIENTS, BATCH NORMALIZATION AND LAYER NORMALIZATION"

Pingback: Recurrent Neural Networks (RNN) – Part 1: Basic RNN / Char-RNN – The Neural Perspective Edit

Pingback: Weights Initialization – The Neural Perspective Edit

Pingback: Recurrent Neural Network (RNN) – Part 4: Custom Cells – The Neural Perspective Edit

Pingback: Reinforcement Learning (RL) – Policy Gradients I – The Neural Perspective Edit

Pingback: The Neural Perspective Edit

LEAVE A REPLY

Enter your comment here...