Table of contents

MLOps Blog

Vanishing and Exploding Gradients in Neural Network Models: Debugging, Monitoring, and Fixing

Katherine (Yi) Li

10 min

7th August, 2023

ML Model Development

Neural network models are trained by the optimization algorithm of **gradient descent**. The input training data helps these models learn, and the loss function gauges how accurate the prediction performance is for each iteration when parameters get updated. As training goes, the goal is to reduce the loss function/prediction error by adjusting the parameters iteratively. Specifically, the gradient descent algorithm has a forward step and a backward step, which lets it do this.

- In forward propagation, input vectors/data move forward through the network using a formula to compute each neuron in the next layer. The formula consists of input/output, activation function *f*, weight *W* and bias *b*:

$$O^l = f(WO^{l-1} + b)$$

This computation iterates forward until it reaches an output or prediction. We then calculate the difference defined by a loss function, e.g., Mean Squared

Error MSE, between the target variable y (in the output layer) and each prediction, y cap:

Table of contents

$$Loss = \frac{1}{n}\sum_{i=1}(yi - \hat{yi})^2$$

- With this initial evaluation, we go through a backward pass (a.k.a. backpropagation) to adjust the weights and biases for each neuron in each layer. To update our neural nets, we first calculate the gradients, which is nothing but the derivatives of the loss function *w.r.t.* weights and biases. Then we nudge our algorithm to take a gradient descent step to minimize the loss function (where alpha is the learning rate):

$$Wnew = Wold - \alpha(\frac{\partial Loss}{\partial Wold})$$

Two opposite scenarios could happen in this case: the derivative term gets extremely small, i.e., approaches zero vs. this term gets extremely large and overflows. These issues are referred to as the Vanishing and Exploding Gradients, respectively.

When you train your model for a while and the performance doesn't seem to get better, chances are your model is suffering from either **vanishing or exploding gradients**.

This article is designated to these issues, and specifically, we will be covering:

- Intuition behind vanishing and exploding gradients problems
- Why these gradients issues happen
- How to identify the gradients issues as the model training goes
- Case demonstrations and solutions to address vanishing and exploding gradients
    - Vanishing gradients
        - ReLU as the activation function
        - Reduce the model complexity
        - Weight initializer with variance
        - Better optimizer with a well-tuned learning rate
    - Exploding gradients

- Gradients clipping
- Proper weight initializer

Table of contents

Check also

Adversarial Attacks on Neural Networks: Exploring the Fast Gradient Sign Method

Vanishing or exploding gradients – intuition behind the problem

Vanishing

During backpropagation, the calculation of (partial) *derivatives/gradients* in the

$$\frac{\partial Loss}{\partial W^l} = \frac{\partial Loss}{\partial O^l}\frac{\partial O^l}{\partial z^l}\frac{\partial z^l}{\partial W^l}$$

where

$$z^l = W^l * O + b$$

As the gradients frequently become SMALLER until they are close to zero, the new model weights (of the initial layers) will be virtually identical to the old weights without any updates. As a result, the gradient descent algorithm never

converges to the optimal solution. This is known as the problem of vanishing gradients, and it's one example of unstable behaviors of neural nets.

Table of contents

On the contrary, if the gradients get LARGER or even NaN as our backpropagation progresses, we would end up with exploding gradients having big weight updates, leading to the divergence of the gradient descent algorithm.

Why vanishing or exploding gradients problem happens?

With this intuitive understanding of what vanishing/exploding gradients are, you must be wondering – why do gradients vanish or explode in the first place, i.e., why do these gradient values diminish or blow up in their travel back through the network?

Vanishing

Simply put, the vanishing gradients issue occurs when we use the Sigmoid or Tanh **activation functions** in the hidden layer; these functions squish a large input space into a small space. Take the Sigmoid as an example, we have the following p.d.f.:

$$f(x) = \frac{1}{1 - e^{-x}}$$

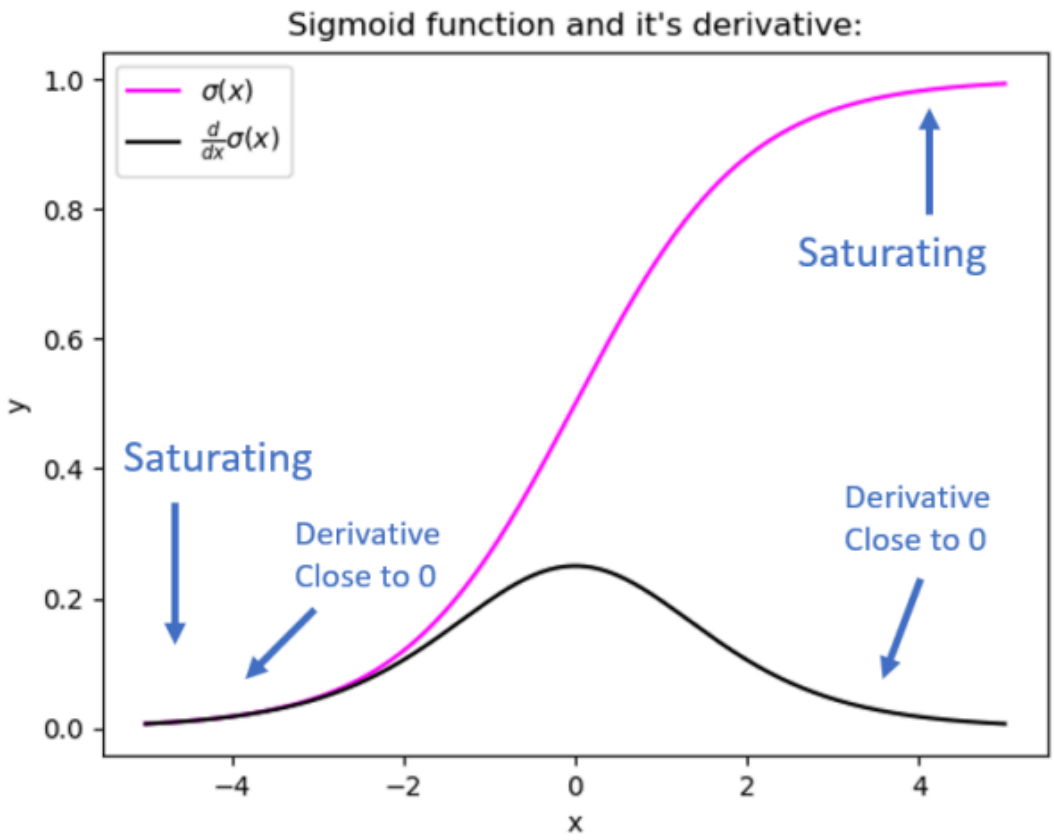
Taking the derivative *w.r.t.* the parameter x, we get:

$$y = \frac{1}{1+e^{-x}}$$

Table of contents

$$= \frac{1}{1+e^{-x}}\left(1 - \frac{1}{1+e^{-x}}\right) = y(1-y)$$

and if we visualize the Sigmoid function and its derivative:



Sigmoid function and its derivative | Source: Author

We can see that the Sigmoid function squeezes our input space into a range between [0,1], and when the inputs become fairly small or fairly large, this function *saturates* at 0 or 1. These regions are referred to as 'saturating regions', whose derivatives become extremely close to zero. The same applies to the Tanh function that *saturates* at -1 and 1.

Suppose that we have inputs that lie in any of the saturating regions, we would essentially have no gradient values to propagate back, leading to a zero update

with just a couple of layers, however, when we add more layers, vanishing gradients in initial layers will result in model training or convergence failure.

Table of contents

decreases exponentially with *n* while the early layers train very slowly and thus the performance of the entire network degrades.

Exploding

Moving on to the exploding gradients, in a nutshell, this problem is due to the **initial weights** assigned to the neural nets creating large losses. Big gradient values can accumulate to the point where large parameter updates are observed, causing gradient descents to oscillate without coming to global minima.

What's even worse is that these parameters can be so large that they overflow and return NaN values that cannot be updated anymore.

How to identify a vanishing or exploding gradients problem?

Acknowledging that the gradients' issues are something we need to avoid or fix when they do happen, how should we know that a model is suffering from vanishing or exploding gradients issues? Following are the few signs.

Vanishing

- Large changes are observed in parameters of later layers, whereas parameters of earlier layers change slightly or stay unchanged
- In some cases, weights of earlier layers can become 0 as the training goes
- The model learns slowly and often times, training stops after a few iterations
- Model performance is poor

Exploding

- Contrary to the vanishing scenario, exploding gradients shows itself as unstable, large parameter changes from batch/iteration to batch/iteration
- Model weights can become NaN very quickly

Table of contents

Practices to fix a vanishing or exploding gradients problem

With these indicators of the gradients problems in mind, let's explore the potential remedies to fix them.

- First, we will be focusing on the **vanishing scenario**: simulating a **binary classification** network model that suffers from this issue, and then demonstrating various solutions to fix it
- By the same token, we will be addressing the **exploding scenario** with a **regression** network model later

By solving different types of deep learning tasks, my goal is to demonstrate different scenarios for you to take away. Please also note that this article is dedicated to providing you with practical approaches and tips, so we will only discuss some intuitions behind each methodology and skip the math or theoretical proofs.

Since observation is a critical part of identifying these issues as discussed above, we will use neptune.ai to track our modeling pipeline:

```
import neptune
import os
myProject = 'YourUserName/YourProjectName'
project =
neptune.init(api_token=os.getenv('NEPTUNE_API_TOKEN'),
                        project=myProject)
project.stop()
```

Solutions for when gradients vanish

- Log the gradients and weights:

Table of contents

```
                    recordWeight=True, npt_exp=None):
    dataGrad, dataWeight = {}, {}
    ## batch update 'weights'
    for wgt, grad in zip(wgts, grads):
        if '/kernel:' not in wgt.name:
            continue
        layerName = wgt.name.split("/")[0]
        dataGrad[layerName] = grad.numpy()
        dataWeight[layerName] = wgt.numpy()
        ## Log in Neptune
        if npt_exp:

npt_exp[f'MeanGrads{layerName.upper()}'].log(np.mean(grad.n
umpy()))

npt_exp[f'MeanWgtBatch{layerName.upper()}'].log(np.mean(wgt
.numpy()))


  gradHist.append(dataGrad)
  lossHist.append(lossVal.numpy())
  if recordWeight:
      wgtsHist.append(dataWeight)
```

- Train the model and use `*tensorflow.GradientTape*` to track and calculate gradients:

```
def fitModel(X, y, model, optimizer,
             n_epochs=n_epochs, curBatch_size=batch_size,
npt_exp=None):

    lossFunc = tf.keras.losses.BinaryCrossentropy()
    subData = tf.data.Dataset.from_tensor_slices((X, y))
    subData =
subData.shuffle(buffer_size=42).batch(curBatch_size)

    gradHist, lossHist, wgtsHist = [], [], []

    for epoch in range(n_epochs):
        print(f'== Starting epoch {epoch} ==')
        for step, (x_batch, y_batch) in enumerate(subData):
```

```
            with tf.GradientTape() as tape:
                ## Predict with the model and calculate
loss
```

Table of contents

```
            ## Calculate gradients using tape and update
the weights
            grads = tape.gradient(lossVal,
model.trainable_weights)
            wgts = model.trainable_weights
            optimizer.apply_gradients(zip(grads,
model.trainable_weights))

            ## Save the Interaction#5 from each epoch
            if step == 5:
                getBatchGradWgts(gradMist=gradMist,
lossMist=lossMist, wgtsMist=wgtsMist,
                                 grads=grads, wgts=wgts,
lossVal=lossVal, npt_exp=npt_exp)
                if npt_exp:
                    npt_exp['BatchLoss'].log(lossVal)

    getBatchGradWgts(gradMist=gradMist, lossMist=lossMist,
wgtsMist=wgtsMist,
                     grads=grads, wgts=wgts,
lossVal=lossVal, npt_exp=npt_exp)
    return gradMist, lossMist, wgtsMist
```

- Visualize the mean gradients from each layer:

```
def gradientsVis(curGradMist, curLossMist, modelName):
    fig, ax = plt.subplots(1, 1, sharex=True,
constrained_layout=True, figsize=(7,5))
    ax.set_title(f"Mean gradient {modelName}")
    for layer in curGradMist[0]:
        ax.plot(range(len(curGradMist)),
[gradList[layer].mean() for gradList in curGradMist],
label=f'Layer_{layer.upper()}')
    ax.legend()
    return fig
```

Now, we will simulate a dataset and build our baseline **binary** classification neural nets:

Table of contents

```
X, y = make_moons(n_samples=3000, shuffle=True ,
noise=0.25, random_state=1234)
```

```
batch_size, n_epochs = 32, 100
```

```
npt_exp = neptune.init(
        api_token=os.getenv('NEPTUNE_API_TOKEN'),
        project=myProject,
        name='VanishingGradSigmoid',
        description='Vanishing Gradients with Sigmoid
Activation Function',
        tags=['vanishingGradients', 'sigmoid', 'neptune'])

    ## Define Neptune callback
    neptune_cbk = NeptuneCallback(run=npt_exp,
base_namespace='metrics')
    def binaryModel(curName, curInitializer, curActivation,
x_tr=None):
        model = Sequential()
        model.add(InputLayer(input_shape=(2, ),
name=curName+"0"))
        model.add(Dense(10,
kernel_initializer=curInitializer,
activation=curActivation, name=curName+"1"))
        model.add(Dense(10,
kernel_initializer=curInitializer,
activation=curActivation, name=curName+"2"))
        model.add(Dense(5,
kernel_initializer=curInitializer,
activation=curActivation,  name=curName+"3"))
        model.add(Dense(1,
kernel_initializer=curInitializer, activation='sigmoid',
name=curName+"4"))
        return model

    curOptimizer = tf.keras.optimizers.RMSprop()
    curInitializer = RandomUniform(-1, 1)
    ## Compile the model
```

```
    model = binaryModel(curName="SIGMOID",
curInitializer=curInitializer, curActivation="sigmoid")
    model.compile(optimizer=curOptimizer,
```

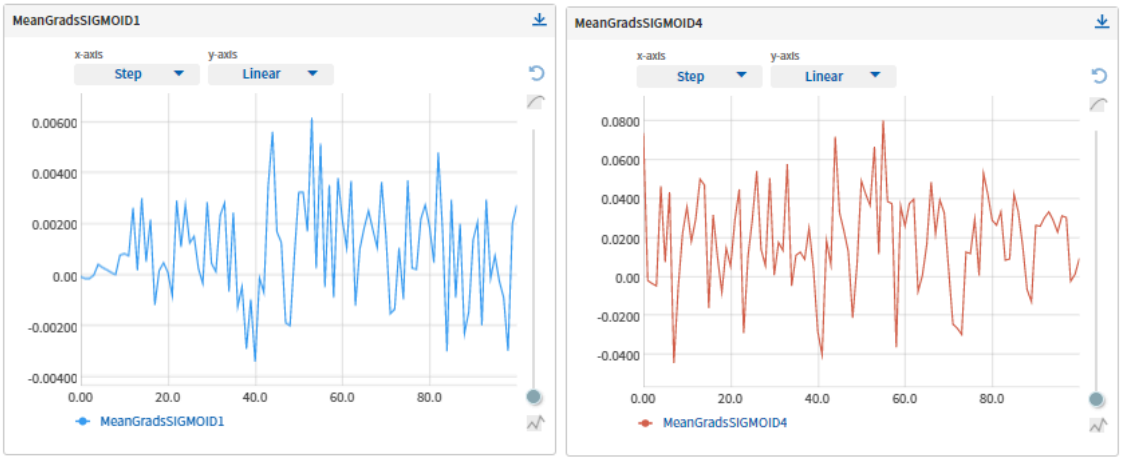
Table of contents

```
model, optimizer=curOptimizer, npt_exp=npt_exp)
    ## log in the plot comparing all layers
    npt_exp['Comparing All
Layers'].upload(neptune.types.File.as_image(gradientsVis(cu
rGradMist, curLossMist, modelName='Sigmoid_Raw')))
    npt_exp.stop()
```

A couple of notes:

- Our current vanilla/baseline model consists of 3 hidden layers, each of which has a sigmoid activation
- We use RMSprop as the optimizer and Uniform [-1, 1] as the weight initializer

Running this model returns the (average) gradients for each layer over all epochs in neptune.ai, and below shows a comparison between Layer 1 and Layer 4:



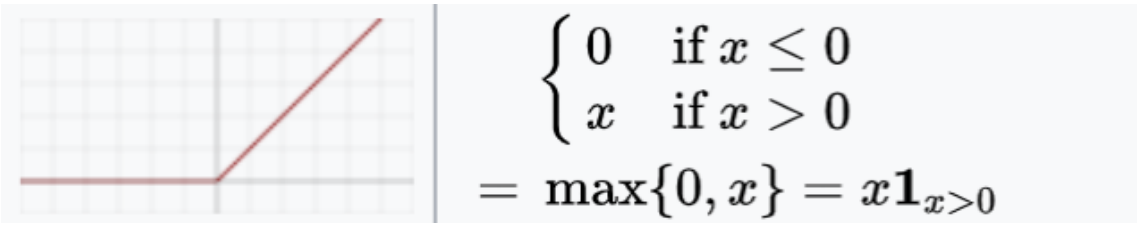
Comparison between Layer 1 and Layer 4 from the baseline Sigmoid model generated using neptune.ai | Source

For Layer 4, we see clear fluctuations in mean gradients as the training proceeds, however, for Layer 1, the gradients are virtually zero, i.e., values approximately less than 0.006. Vanishing gradients happened! Now let's talk about how we can fix this.

Use ReLU as the activation function

As aforementioned, the vanishing gradients problem is due to the saturating nature of the Sigmoid or Tanh function. Hence, an effective remedy would be to switch to other activation functions that are **non-saturated** for their derivative.

Table of contents



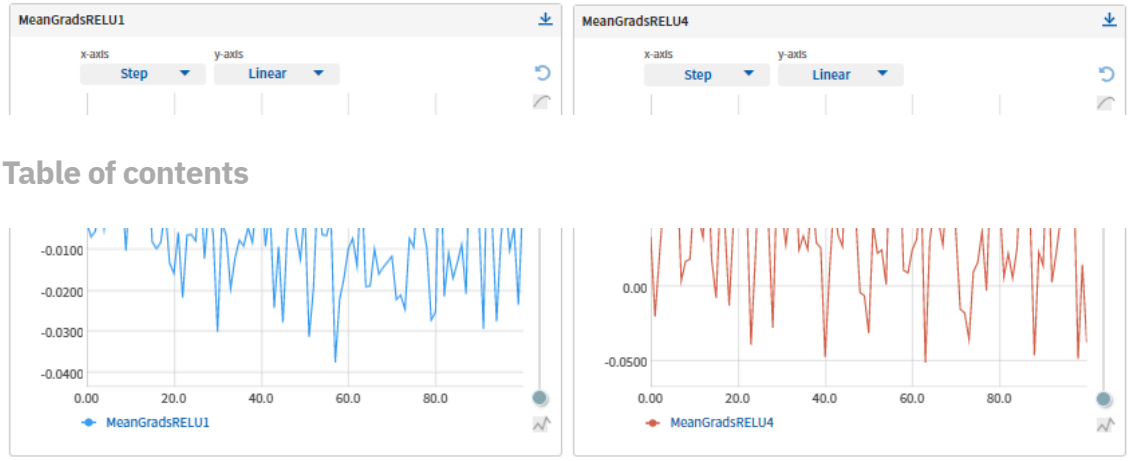
ReLU as the activation function | Source

As shown in this graph, ReLU doesn't saturate for positive input x. When x <= 0, ReLU has derivative/gradient = 0, and when x > 0, the derivative/gradient = 1. Therefore, multiplying ReLU derivatives returns either 0 or 1; thus, there won't be vanishing gradients.

To implement the ReLU activation, we can simply specify `*relu*` in our model function shown below:

```
## Compile the model
model = binaryModel(curName="Relu",
curInitializer=curInitializer, curActivation="relu")

model.compile(optimizer=curOptimizer,
loss='binary_crossentropy', metrics=['accuracy'])
```

Running this model and comparing gradients calculated from this model, we observe changes in gradients across epochs even for the first Layer labeled RELU1:



Comparison between Layer 1 and Layer 4 from the ReLu model generated using neptune.ai | Source

In most cases, the ReLU-like activation function itself should be sufficient to handle the vanishing gradients issue. However, does this mean that we should always use ReLU and ditch Sigmoid completely? Well, the fact that vanishing gradients exist shouldn't stop you from using Sigmoid, which has many desirable properties such as being monotonic and easily differentiable. There are approaches to get around this issue even with Sigmoid activation function, and these methods are what we will be experimenting within the following sessions.

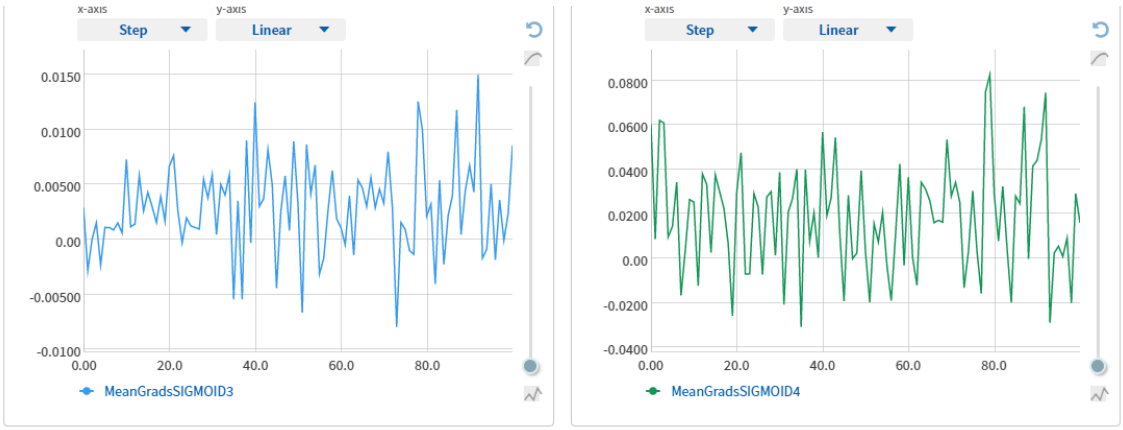
Reduce the complexity of the model

Since the root cause of vanishing gradients lies in multiplication of a bunch of small gradients, intuitively, it makes sense to fix this issue by reducing the number of gradients, i.e., reducing the number of layers in our network. For example, rather than specifying 3 hidden layers as in our baseline model, we can only keep 1 hidden layer to make our model simpler:

```
def binaryModel(curName, curInitializer, curActivation,
x_tr=None):
        model = Sequential()
        model.add(InputLayer(input_shape=(2, ),
name=curName+"0"))
        model.add(Dense(3,
kernel_initializer=curInitializer,
activation=curActivation,  name=curName+"3"))
        model.add(Dense(1,
kernel_initializer=curInitializer, activation='sigmoid',
name=curName+"4"))
        return model
```

This model gives us clear gradients updates showing in this plot:

Table of contents



Comparison between Layer 1 and Layer 4 from the reduced Sigmoid model generated using neptune.ai | Source

One caveat of this approach is that our model performance may not be as good as more complex models (with more hidden layers).

Use weight initializer with variance

When our initial weights are set too small or lacking variance, it often will cause gradients to vanish. Recall that in our baseline model, we initialized our weights as a uniform [-1, 1] distribution, this may fall into the pitfall that these weights are too small!

In their 2010 paper, Xavier Glorot and Yoshua Bengio provided theoretical justification of sampling the initial weights from a uniform or normal distribution of **certain variances**, and maintaining the variance of activations the same across all layers.

In Keras/Tensorflow, this methodology is implemented as the Glorot Normal `*glorot_normal*` and Glorot Uniform `*glorot_uniform*`, which as the names suggest, samples initial weights from a (truncated) normal and uniform distribution, respectively. Both take into consideration the number of input and output units.

For our model, let's experiment with the *glorot_uniform*, which, according to Keras documentation:

> Draws samples from a uniform distribution within `[-limit, limit]`, where `limit = sqrt(6 / (fan_in + fan_out))` (`fan_in` is the number of input units in the weight tensor and `fan_out` is the number of output units).

Going back to the original model with 3 hidden layers, we initialize model weights as *glorot_uniform*:

Table of contents

```
x_tr=None):
        model = Sequential()
        model.add(InputLayer(input_shape=(2, ),
name=curName+"0"))
        model.add(Dense(10,
kernel_initializer=curInitializer,
activation=curActivation, name=curName+"1"))
        model.add(Dense(10,
kernel_initializer=curInitializer,
activation=curActivation, name=curName+"2"))
        model.add(Dense(5,
kernel_initializer=curInitializer,
activation=curActivation,  name=curName+"3"))
        model.add(Dense(1,
kernel_initializer=curInitializer, activation='sigmoid',
name=curName+"4"))
        return model

    curOptimizer = tf.keras.optimizers.RMSprop()
    optimizer = curOptimizer
    ### Weight needs variance
    curInitializer = 'glorot_uniform'
    ## log in the plot comparing all layers
    npt_exp['Comparing All
Layers'].upload(neptune.types.File.as_image(gradientsVis(cu
rGradHist, curLossHist,

modelName='Sigmoid_NormalWeightInit')))
    npt_exp.stop()
```

Checking our neptune.ai tracker, we see gradients change with this weight initialization, although Layer 1 (on the left) shows less of a fluctuation as compared to the last layer:

Table of contents

Comparison between Layer 1 and Layer 4 from the Glorot weight initializer model generated using neptune.ai | Source

Select better optimizer and adjust learning rate

Now, we have tackled the derivatives and the selection of initial weights, the last piece in the formula is learning rate. With gradients approaching zero, the optimizer gets trapped in sub-optimal local minima or saddle point. To overcome this challenge, we can employ an optimizer with a momentum that factors in the accumulated previous gradients. For example, Adam has a momentum term calculated as the exponentially decaying average of the past gradients.

In addition, as an efficient optimizer, Adam can converge or diverge quickly. Hence, slightly reducing the learning rate will help prevent your network from diverging too easily, thus reducing the possibility of gradients approaching zero. To use the Adam optimizer, all we need to modify is the `*curOptimizer*` arg.:

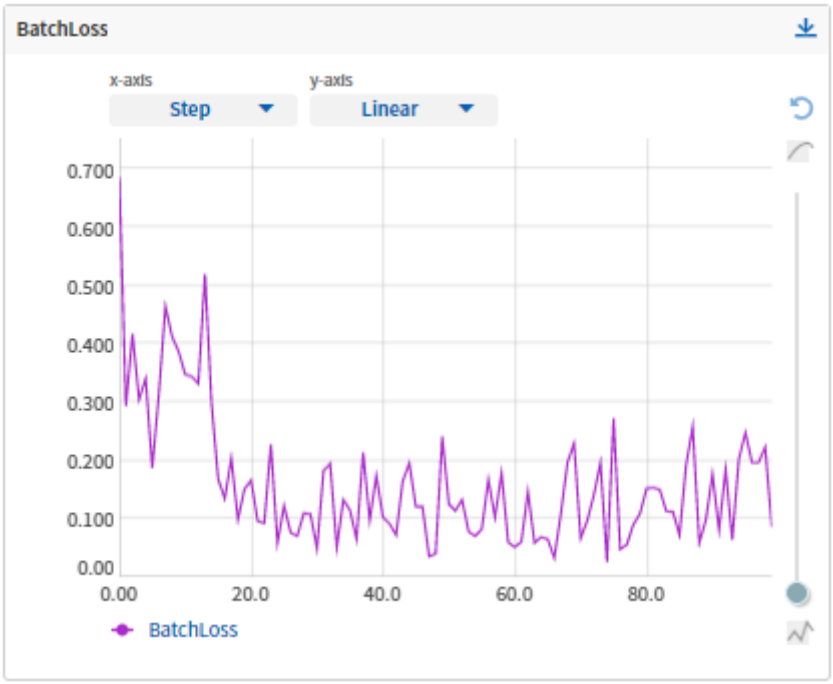
```
curOptimizer = keras.optimizers.Adam(learning_rate=0.008)
## reduce the learning rate with Adam

curInitializer = RandomUniform(-1, 1)
## Compile the model
model = binaryModel(curName="SIGMOID",
curInitializer=curInitializer, curActivation="sigmoid")
model.compile(optimizer=curOptimizer,
loss='binary_crossentropy', metrics=['accuracy'])
```

In the code above, we specified Adam as the model optimizer along with a

is the comparison of Layer 1 and Layer 4 gradients:

Comparison between Layer 1 and Layer 4 from the Adam model generated using neptune.ai | Source

As we can see, with Adam and a well-tuned small learning rate, we see variations in the gradients taking their values away from zero, and our model also gets converged to a local minima based on the loss plot shown below:



Selecting better optimizer and adjusting learning rate | Source

Up until this point, we have walked through solutions for vanishing gradients, let's move on to the exploding gradients issue.

Solutions for when gradients explode

For the exploding gradients issue, let's take a look at this regression model.

```
# Generate regression dataset
nfeatures = 15
```

Table of contents

```
# Define the regression model
def regressionModel(X, y, curInitializer, USE_L2REG,
secondLayerAct='relu'):
    ## Construct the neural nets
    inp = Input(shape = (X.shape[1],))
    if USE_L2REG:
        ## need to change activation function as well
        x = Dense(35, activation='tanh',
kernel_initializer=curInitializer,
                  kernel_regularizer=regularizers.l2(0.01),

activity_regularizer=regularizers.l2(0.01))(inp)
    else:
        x = Dense(35, activation=secondLayerAct,
kernel_initializer=curInitializer)(inp)

    out = Dense(1, activation='linear')(x)
    model = Model(inp, out)
    return model
```

To compile the model, we will use a Uniform [4, 5] weight initializer along with ReLu activation, for the purpose of creating an exploding gradients situation:

```
sgd = tf.keras.optimizers.SGD()
curOptimizer = sgd

#### Uniform init
curInitializer = RandomUniform(4,5)

model = regressionModel(X, y, curInitializer,
USE_L2REG=False)
model.compile(loss='mean_squared_error',
optimizer=curOptimizer, metrics=['mse'])

curModelName = 'Relu_Raw'
```

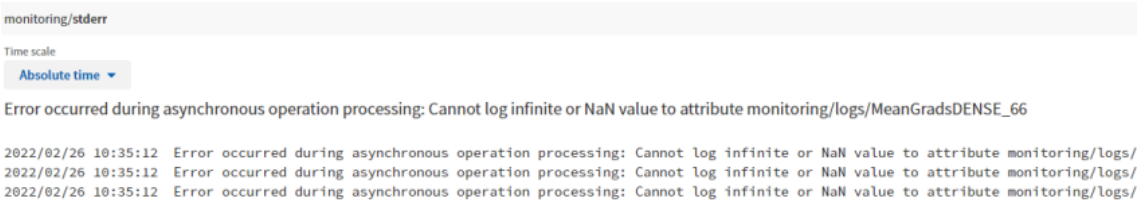
```
## Train and Log in Neptune
curGradHist, curLossHist, curWgtHist = fitModel(X, y,
```

```
model, optimizer=curOptimizer,
modelType = 'regression',
npt_exp=npt_exp)
```

Table of contents

```
rGradMist, curLossMist,

modelName=curModelName)))
npt_exp.stop()
```

Having this big of a weight initialization, it comes to no surprise that as the training goes, the following error message shows up in our neptune.ai tracker, which, as discussed before, clearly indicates that our gradients exploded:



Error message in neptune.ai | Source

Gradients clipping

To prevent gradients from exploding, one of the most effective ways is gradient clipping. In a nutshell, gradient clipping caps the derivatives to a threshold and uses the capped gradients to update the weights throughout. If you are interested in a detailed explanation of this method, please refer to the article “Understanding Gradient Clipping (and How It Can Fix Exploding Gradients Problem)”.

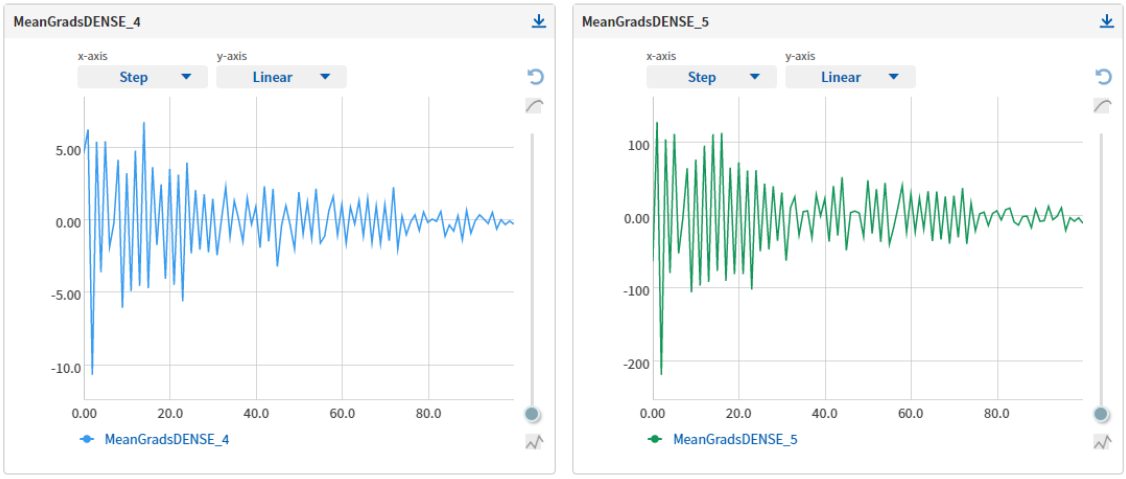
Capping the gradients to a certain value can be specified by the `*clipvalue*` arg. as shown below:

```
### Gradients clipping
sgd = tf.keras.optimizers.SGD(clipvalue=50)
curOptimizer = sgd
curInitializer = 'glorot_normal'

model = regressionModel(X, y, curInitializer,
USE_L2REG=False)
model.compile(loss='mean_squared_error',
```

```
optimizer=curOptimizer, metrics=['mse'])
curModelName = 'GradClipping'
```

Table of contents

the defined range:



Layer gradients from the gradients clipping model generated using neptune.ai | Source

Proper weight initializer

As aforementioned, one primary cause of gradients exploding lies in too large of a weight initialization and update, and this is the reason why gradients in our regression model exploded. Hence, initializing model weights properly is the key to fix this exploding gradients problem.

Same as the vanishing gradients, we will implement the Glorot initialization with a normal distribution.

Since the Glorot initialization works the best with Tanh or Sigmoid, we will specify Tanh as the activation function in this experiment:

```
curOptimizer = tf.keras.optimizers.SGD()
## Glorot init
curInitializer = 'glorot_normal'

## Tanh as the activation function
model = regressionModel(X, y, curInitializer,
USE_L2REG=False, secondLayerAct='tanh')

model.compile(loss='mean_squared_error',
```

```
optimizer=curOptimizer, metrics=['mse'])

curModelName = 'GlorotInit'
```

Table of contents

Here is the gradients plot from this model, which fixed the exploding gradients issue:

Layer gradients from the Glorot initialization model generated using neptune.ai | Source

L2 norm regularization

In addition to weight initialization, another excellent approach is employing L2 regularization, which penalizes large weight values by imposing a squared term of model weights to the loss function:

$$Loss = Error(Y - \hat{Y}) + \lambda \sum_{1}^{n} w_i^2$$

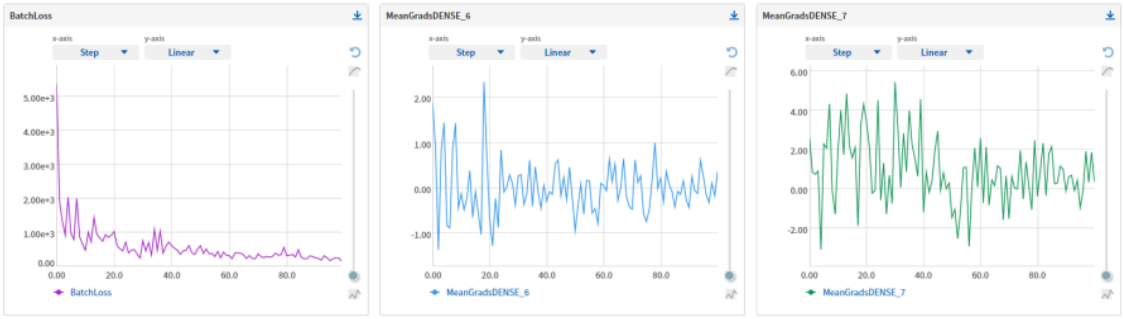
Adding the L2 norm oftentimes will result in smaller weight updates throughout the network, and implementing this regularization in Keras is rather straightforward with the args. `*kernal_regularizer*` and `*activity_regularizer*`:

```
curInitializer = 'glorot_normal'
x = Dense(35, activation='tanh',
kernel_initializer=curInitializer,
          kernel_regularizer=regularizers.l2(0.01),
          activity_regularizer=regularizers.l2(0.01))(inp)
```

```
### Using the regressionModel function
curInitializer = 'glorot_normal'
model = regressionModel(X, y, curInitializer,

curModelName = 'L2Reg'
```

Table of contents

We set the initial weights of this model as glorot normal, following the recommendation from Razvan etc., 2013 of initializing parameters with small values and variance. Here shows the loss curve and the gradients for each layer:



Layer gradients from the L2 regularization model generated using neptune.ai | Source

Again, the gradients are controlled within a reasonable range through all epochs, and the model gradually converges.

Check also

Fighting Overfitting With L1 or L2 Regularization: Which One Is Better?

Final words

In addition to the main techniques we discussed in this article, other methods worth trying to avoid/fix the gradients include **Batch Normalization** and **Scaling the input data**. Both methods can make your network more robust.

The intuition is that during backprop, our input data of each layer can vary tremendously (as the output from the previous layer). Using batch

normalization allows us to fix the mean and variance of the input data for each layer, and thus prevent it from shifting around too much. With a more robust network, it will be less likely to come across the two gradients issues.

Table of contents

network training – the vanishing and exploding gradients problems. We explained their causes and consequences. We also walked through various approaches to address the two problems.

Hope you have found this article useful and learned practical techniques to use in training your own neural network models. For your reference, the full code is available in my GitHub repo here and the Neptune project is available here.

Was the article useful?

Time Series

Table of contents

Newsletter

Top MLOps articles, case studies, events (and more) in your inbox every month.

Your e-mail

Get Newsletter

PRODUCT

SOLUTIONS

DOCUMENTATION

Table of contents

COMMUNITY

COMPANY

MLOps: What, Why, and How MLOps Landscape: Top Tools
Building an ML Platform ML Experiment Tracking: What, Why, and How
How to Build an Experiment Tracker