Understanding and Addressing Vanishing and Exploding Gradients in Deep Learning

Machine Learning Tutorial

Nazmul Hossain
23015862

https://github.com/nazmul-nil/vanishing_gradients_tutorial_MLNN

Sections

❑ Introduction

❑ Why Do Gradients Matter?

❑ What Are Vanishing and Exploding Gradients?

❑ Diagnosing the Problem with Gradient Logger

Sections

- ❑ Solutions to Vanishing and Exploding Gradients
- ❑ Best Practices for Avoiding Gradient Issues
- ❑ Loss Curves for Different Techniques
- ❑ Loss Reduction Over Epochs

Sections

❑ Key Insights

❑ Conclusion

❑ References

Introduction

What are vanishing and exploding gradients?

- Vanishing gradients occur when the gradients (the values used to update weights in a neural network) become very small as they are propagated backward through the network. This causes earlier layers to learn very slowly or stop learning altogether (Hochreiter et al., 1997).
- Exploding gradients occur when gradients become excessively large during backpropagation, destabilizing the learning process and often resulting in numerical overflow (Bengio et al., 1994).

Introduction

Why are they important?

- Both problems are significant barriers to training deep neural networks effectively. They hinder the optimization process, causing the model to either fail to learn or produce unstable outputs (Hochreiter et al., 1997).
- Solving these issues ensures faster, more reliable convergence and enables the effective training of very deep architectures.

Why Do Gradients Matter

Role of gradients in deep learning:

- Gradients are numerical values computed during backpropagation that indicate how much a model's weights should change to reduce the loss (Rumelhart et al., 1986).
- Without gradients, the optimization process (learning) cannot occur.

Why Do Gradients Matter

Impact of vanishing/exploding gradients:

- Vanishing gradients cause earlier layers in the network to stop learning. This is especially problematic in deep networks where these layers are critical for feature extraction (Hochreiter et al., 1997).
- Exploding gradients lead to excessively large weight updates, causing the model to diverge or produce `NaN` (not a number) values during training (Bengio et al., 1994).

What Are Vanishing and Exploding Gradients?

Vanishing Gradients:

- Vanishing gradients occur when small gradient values are propagated backward through the network during backpropagation. This is common with activation functions like sigmoid and tanh, which squash large input values into a small range, leading to small derivatives (Hochreiter et al., 1997).
- With deep networks, this problem is compounded because the gradient values are repeatedly multiplied by small numbers, causing them to diminish exponentially (Hochreiter et al., 1997).

What Are Vanishing and Exploding Gradients?

Exploding Gradients:

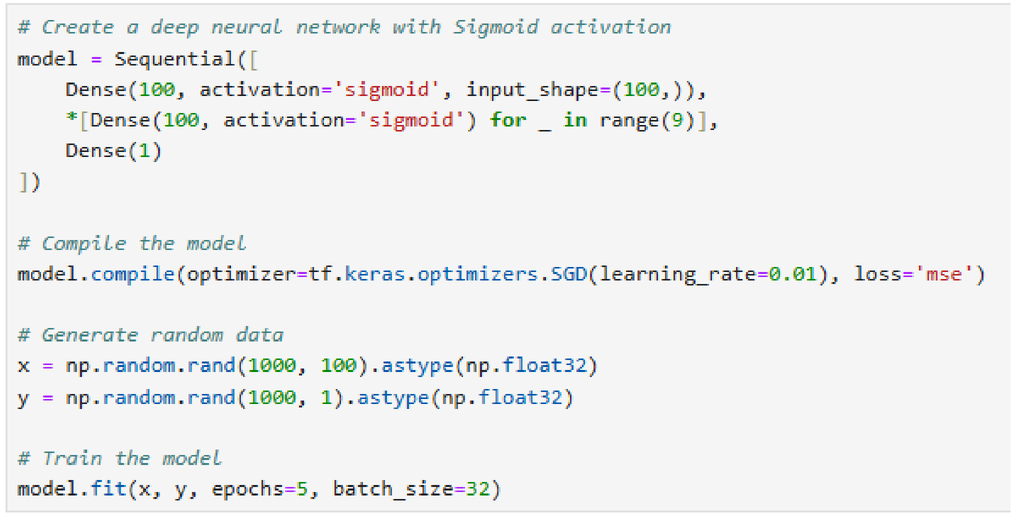
- Exploding gradients occur when the network's weights are initialized with large values or when numerical instabilities amplify the gradient during backpropagation. This results in exponentially increasing gradient values that cause the loss to diverge (Bengio et al., 1994).
- This issue is particularly severe in recurrent neural networks (RNNs) or very deep feedforward networks (Pascanu et al., 2013).

What Are Vanishing and Exploding Gradients?

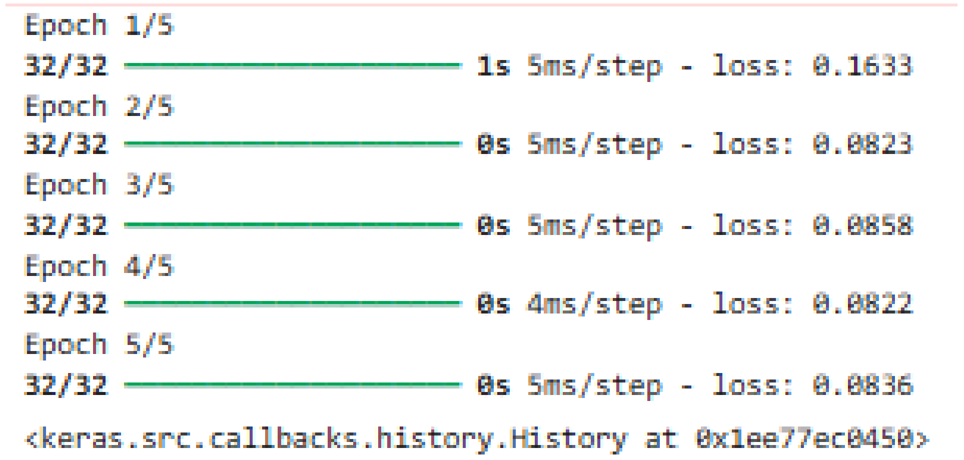
Key Impacts:

- Vanishing gradients prevent earlier layers in the network from learning effectively, resulting in poor feature extraction.
- Exploding gradients destabilize the training process and often cause the model to fail completely.

What Are Vanishing and Exploding Gradients?

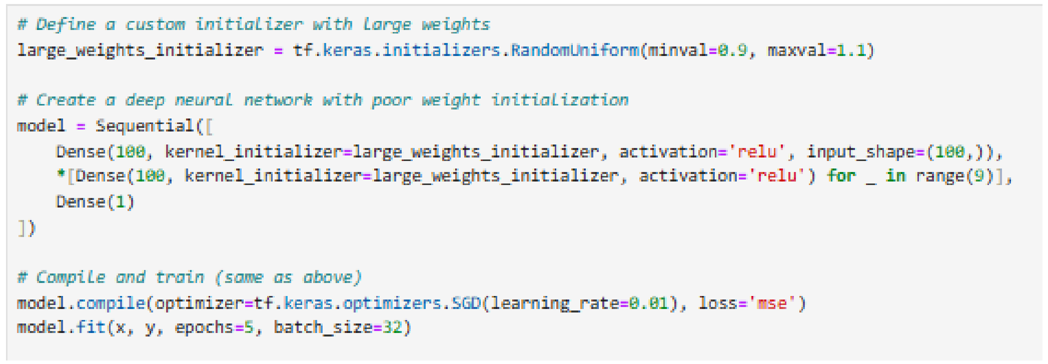


```
# Create a deep neural network with Sigmoid activation
model = Sequential([
    Dense(100, activation='sigmoid', input_shape=(100,)),
    *[Dense(100, activation='sigmoid') for _ in range(9)],
    Dense(1)
])

# Compile the model
model.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=0.01), loss='mse')

# Generate random data
x = np.random.rand(1000, 100).astype(np.float32)
y = np.random.rand(1000, 1).astype(np.float32)

# Train the model
model.fit(x, y, epochs=5, batch_size=32)
```

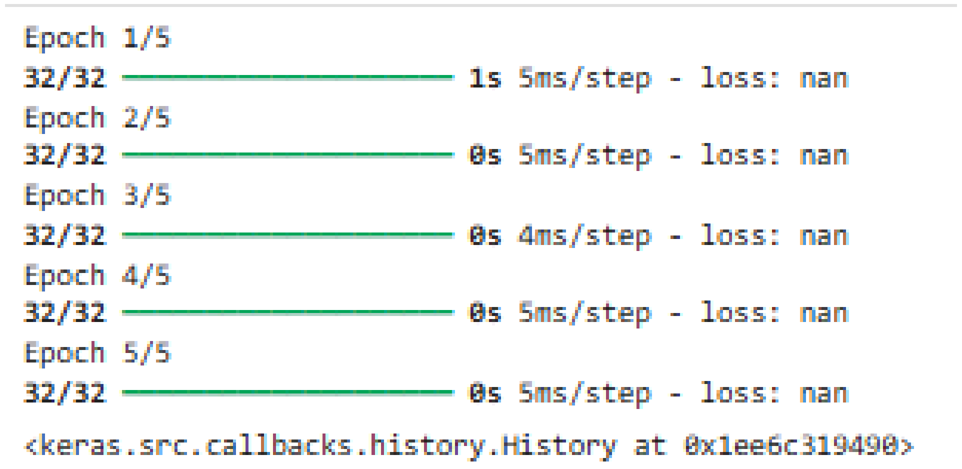


```
Epoch 1/5
32/32 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - loss: 0.1633
Epoch 2/5
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 0.0823
Epoch 3/5
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 0.0858
Epoch 4/5
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.0822
Epoch 5/5
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 0.0836
<keras.src.callbacks.history.History at 0x1ee77ec0450>
```

What Are Vanishing and Exploding Gradients?



```python
# Define a custom initializer with large weights
large_weights_initializer = tf.keras.initializers.RandomUniform(minval=0.9, maxval=1.1)

# Create a deep neural network with poor weight initialization
model = Sequential([
    Dense(100, kernel_initializer=large_weights_initializer, activation='relu', input_shape=(100,)),
    *[Dense(100, kernel_initializer=large_weights_initializer, activation='relu') for _ in range(9)],
    Dense(1)
])

# Compile and train (same as above)
model.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=0.01), loss='mse')
model.fit(x, y, epochs=5, batch_size=32)
```

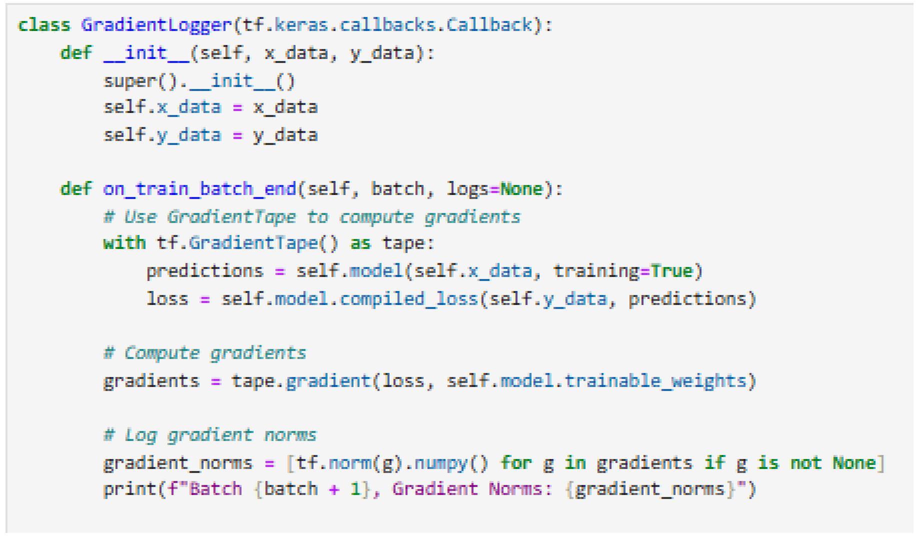


```
Epoch 1/5
32/32 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - loss: nan
Epoch 2/5
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: nan
Epoch 3/5
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: nan
Epoch 4/5
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: nan
Epoch 5/5
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: nan
<keras.src.callbacks.history.History at 0x1ee6c319490>
```

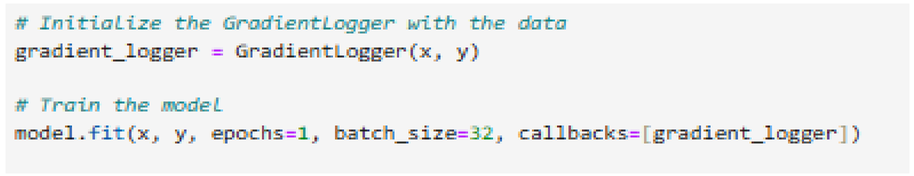
Diagnosing the Problem with Gradient Logger

To check if gradients are vanishing or exploding, we can use a callback to track their size during training. Very small gradients mean vanishing, and very large ones mean exploding (Pascanu et al., 2013).

Small norms like `[0.0001, 0.00005]` show vanishing gradients (Hochreiter et al., 1997), while large ones like `[100, 500]` indicate exploding gradients.



```python
class GradientLogger(tf.keras.callbacks.Callback):
    def __init__(self, x_data, y_data):
        super().__init__()
        self.x_data = x_data
        self.y_data = y_data

    def on_train_batch_end(self, batch, logs=None):
        # Use GradientTape to compute gradients
        with tf.GradientTape() as tape:
            predictions = self.model(self.x_data, training=True)
            loss = self.model.compiled_loss(self.y_data, predictions)

        # Compute gradients
        gradients = tape.gradient(loss, self.model.trainable_weights)

        # Log gradient norms
        gradient_norms = [tf.norm(g).numpy() for g in gradients if g is not None]
        print(f"Batch {batch + 1}, Gradient Norms: {gradient_norms}")
```

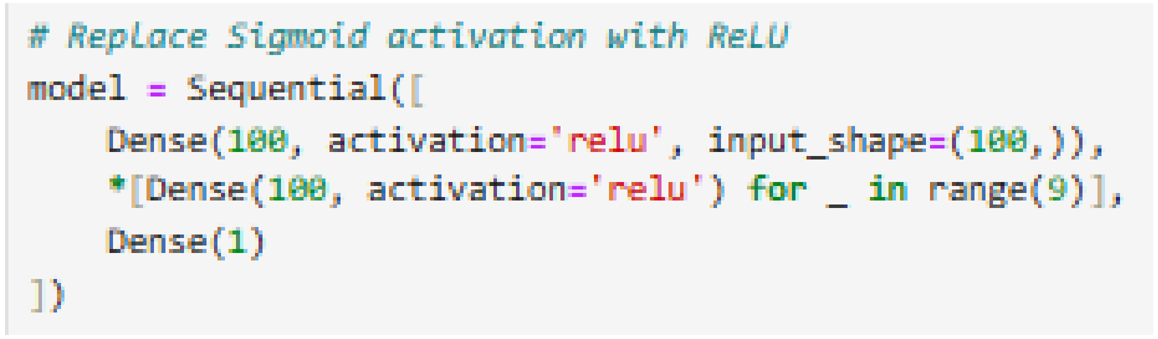


```python
# Initialize the GradientLogger with the data
gradient_logger = GradientLogger(x, y)

# Train the model
model.fit(x, y, epochs=1, batch_size=32, callbacks=[gradient_logger])
```

Solutions to Vanishing and Exploding Gradients

ReLU Activation

Replacing sigmoid or tanh with ReLU can reduce vanishing gradients. ReLU does not squash values into small ranges, so gradients remain larger (Nair and Hinton, 2010).

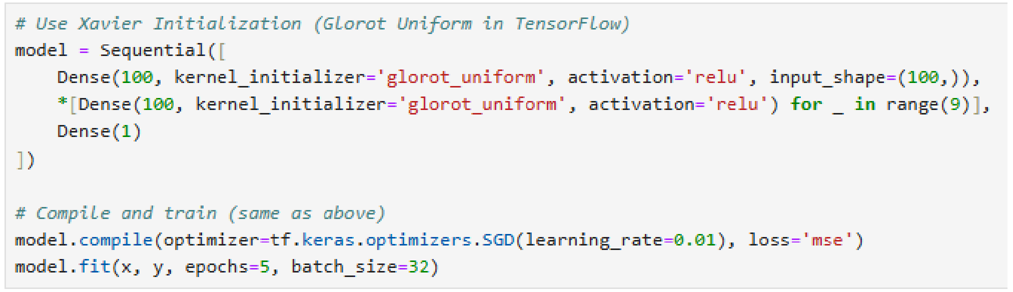


```
# Replace Sigmoid activation with ReLU
model = Sequential([
    Dense(100, activation='relu', input_shape=(100,)),
    *[Dense(100, activation='relu') for _ in range(9)],
    Dense(1)
])
```

Solutions to Vanishing and Exploding Gradients

Weight Initialization

Proper initialization like Xavier (Glorot and Bengio, 2010) ensures gradients are scaled correctly, preventing both vanishing and exploding.

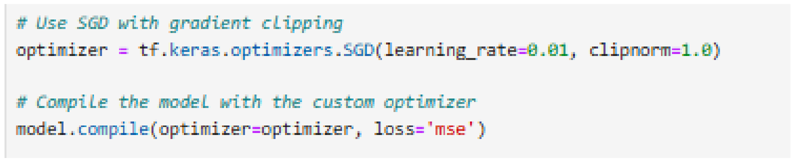


```
# Use Xavier Initialization (Glorot Uniform in TensorFlow)
model = Sequential([
    Dense(100, kernel_initializer='glorot_uniform', activation='relu', input_shape=(100,)),
    *[Dense(100, kernel_initializer='glorot_uniform', activation='relu') for _ in range(9)],
    Dense(1)
])

# Compile and train (same as above)
model.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=0.01), loss='mse')
model.fit(x, y, epochs=5, batch_size=32)
```

Solutions to Vanishing and Exploding Gradients

Gradient Clipping

Clipping caps gradients at a fixed value to prevent them from growing too large (Pascanu et al., 2013).



```
# Use SGD with gradient clipping
optimizer = tf.keras.optimizers.SGD(learning_rate=0.01, clipnorm=1.0)

# Compile the model with the custom optimizer
model.compile(optimizer=optimizer, loss='mse')
```

Solutions to Vanishing and Exploding Gradients

What Happens After Applying Solutions

- **ReLU**: Gradients do not vanish as they propagate back.
- **Weight Initialization**: Ensures gradients are neither too small nor too large.
- **Batch Normalization**: Smooths gradient flow and accelerates learning.
- **Gradient Clipping**: Prevents exploding gradients from destabilizing training.

Best Practices for Avoiding Gradient Issues

Use Proper Activation Functions

- ReLU and its variants (like Leaky ReLU) are good choices for deep networks. These functions allow gradients to flow without shrinking significantly (Nair and Hinton, 2010).
- Avoid sigmoid or tanh activations in deep layers as they compress inputs, causing small gradients (Hochreiter et al., 1997).

Best Practices for Avoiding Gradient Issues

Initialize Weights Correctly

- Use Xavier initialization (Glorot and Bengio, 2010) for balanced gradient scaling in deep feedforward networks.
- Use He initialization for models with ReLU activation to further stabilize training.

Best Practices for Avoiding Gradient Issues

Apply Gradient Clipping

- Gradient clipping caps gradient magnitudes, preventing instability caused by exploding gradients (Pascanu et al., 2013).
- It is especially useful in recurrent neural networks or very deep models.

Loss Curves for Different Techniques

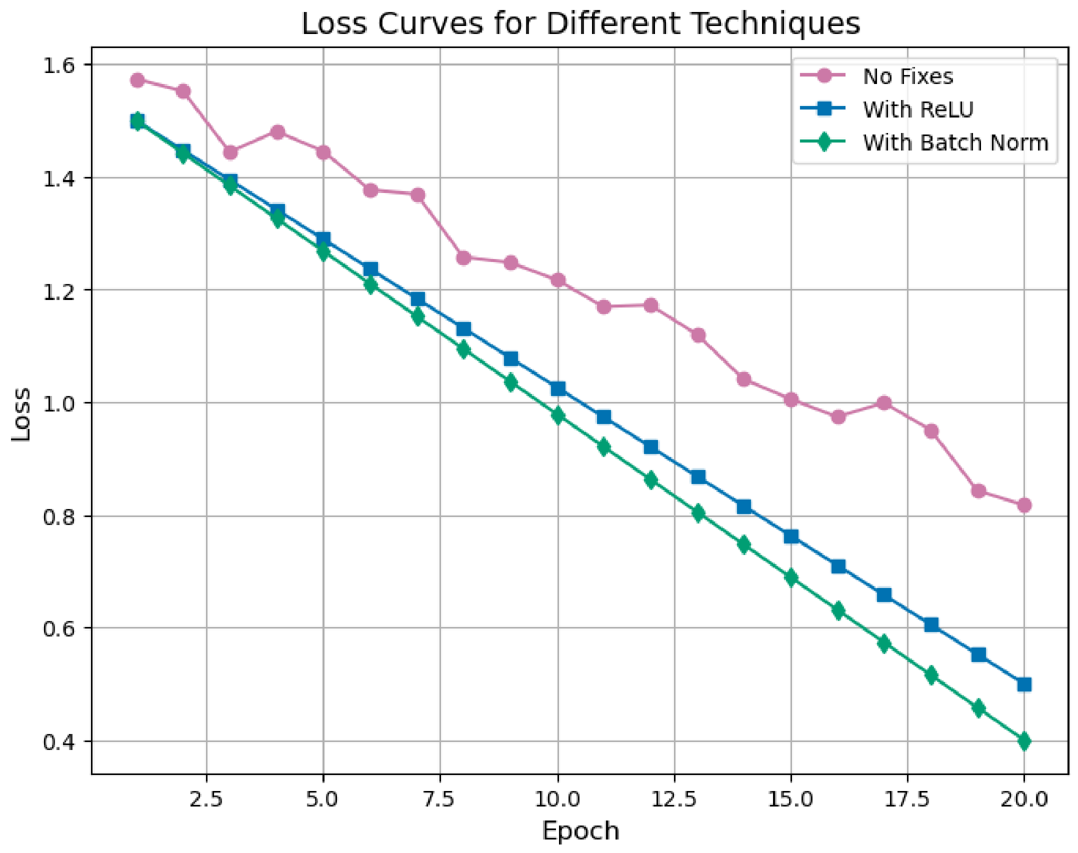
Comparing Techniques to Address Gradient Issues

- This plot compares how different techniques affect the loss reduction during training:
  - **No Fixes:** Training progresses slowly due to vanishing or exploding gradients.
  - **With ReLU:** Loss reduction improves as ReLU prevents vanishing gradients.
  - **With Batch Norm:** The fastest loss reduction, as batch normalization ensures stable gradient flow and faster learning.

Explanation: The use of ReLU and Batch Normalization clearly enhances training efficiency compared to a model with no gradient-related fixes.

Loss Curves for Different Techniques

Comparing Techniques to Address Gradient Issues

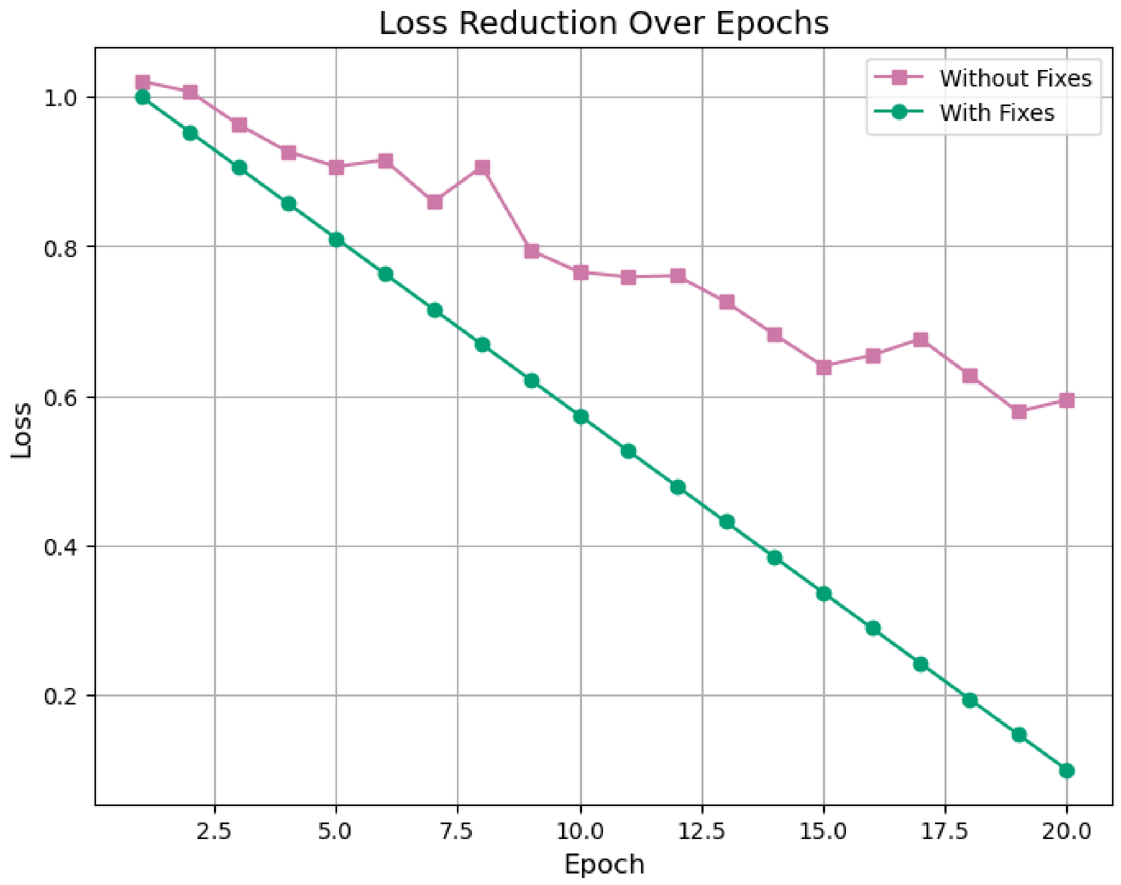


Loss Reduction Over Epochs

Impact of Fixes on Training Stability

- This plot highlights the overall difference between training a model with and without gradient-related fixes:
  - **Without Fixes:** Loss decreases slowly, often stagnating as training progresses.
  - **With Fixes:** Loss decreases steadily, demonstrating smoother and more effective learning.

Explanation: The use of ReLU and Batch Normalization clearly enhances training efficiency compared to a model with no gradient-related fixes.



Loss Reduction Over Epochs

Impact of Fixes on Training Stability

Key Insights

Training deep neural networks often faces challenges due to vanishing and exploding gradients, which can significantly affect learning. Addressing these problems ensures efficient and stable training. Here's a quick recap of the key points:

Vanishing and Exploding Gradients

- **Vanishing gradients**: Gradients become very small as they are propagated backward, causing earlier layers to learn very slowly (Hochreiter et al., 1997).
- **Exploding gradients**: Gradients grow too large, leading to instability in training and divergence of the model (Pascanu et al., 2013).

Key Insights

How to Identify Gradient Issues

- Use a gradient logger to track gradient norms during training.
- Vanishing gradients appear as small norms, while exploding gradients result in excessively large norms (Pascanu et al., 2013).

Key Insights

Solutions

- **ReLU Activation**: Prevents vanishing gradients by avoiding the squashing effect of sigmoid and tanh (Nair and Hinton, 2010).
- **Proper Initialization**: Xavier and He initializations balance gradient scaling, avoiding both issues (Glorot and Bengio, 2010).
- **Batch Normalization**: Stabilizes inputs to each layer, improving gradient flow and speeding up training (Ioffe and Szegedy, 2015).
- **Gradient Clipping**: Caps large gradients to prevent instability caused by exploding gradients (Pascanu et al., 2013).

Conclusion

Deep neural networks are powerful tools, but they come with challenges like vanishing and exploding gradients. These problems can slow down learning, destabilize training, or even prevent convergence. By understanding these issues and applying proven solutions, we can train deep networks efficiently and effectively.

What We Covered

- **What are vanishing and exploding gradients?**
  - Gradients that become too small or too large during backpropagation hinder training.
- **How to diagnose these problems?**
  - Using gradient tracking methods like the `GradientLogger` to monitor gradient sizes.
- **How to solve these issues?**
  - Techniques like ReLU activation, Xavier and He initialization, batch normalization, and gradient clipping.

References

1. Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2), pp.157–166.

2. Hochreiter, S., Bengio, Y., Frasconi, P., and Schmidhuber, J. (1997). Gradient flow in recurrent nets: The difficulty of learning long-term dependencies. *A Field Guide to Dynamical Recurrent Neural Networks*, pp.237–243.

3. Pascanu, R., Mikolov, T., and Bengio, Y. (2013). On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning*, pp.1310–1318.

4. Rumelhart, D.E., Hinton, G.E., and Williams, R.J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088), pp.533–536.

5. Nair, V. and Hinton, G.E. (2010). Rectified linear units improve restricted Boltzmann machines. *Proceedings of the 27th International Conference on Machine Learning (ICML)*, pp.807–814.

6. Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS)*, pp.249–256.

7. Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, pp.448–456.