```
#####################################################################
#                                   #
# CS435 Generative AI: Security, Ethics and Governance      #
#                                   #
# Instructor: Dr. Adnan Masood                  #
# Contact:   adnanmasood@gmail.com                  #
#                                   #
# Notebook is MIT Licensed                  #
#####################################################################
```

Teaching Backpropagation and Vanishing Gradients in Neural Networks

Instructor: Dr. Adnan Masood

Welcome everyone! In this session, we will dive deep into two key concepts in Neural Networks:

1. **Backpropagation** (often shortened to *backprop*)
2. **Vanishing Gradients**

We will explore these topics at **five different levels** (from very simple to more advanced), building from intuitive explanations all the way to code and mock calculations. We'll also provide a **brief history**, **underlying math**, a **step-by-step example** of how to create the technology from scratch, and how it can be used to solve both illustrative and practical, real-world problems. Finally, we'll have a **Q&A section**, some code examples with **TODOs**, and a **Glossary** of terms at the end.

1. Building an Intuitive Understanding

What is Backpropagation?

- Imagine you are learning to shoot basketball hoops. Each time you shoot, you see how far you missed and try to correct your aim. Over time, you adjust your arm's angle, strength, etc. until you improve.
- Similarly, in a neural network, backpropagation is the way a computer learns from mistakes. It sees how "off" its guess was (the miss) and adjusts all the "knobs" (weights and biases) so the next guess is better.

What is Vanishing Gradient?

- If you had to pass a message through a big chain of friends, sometimes the message gets quieter or lost by the time it reaches the last friend. That's what happens in a very deep neural network: the "fix-it" signal (gradient) can get so small (vanish) that the first layers barely learn anything.

Backpropagation is:

- A method to train multi-layer neural networks.
- It uses the *chain rule* from calculus to figure out how to change each weight in the network in order to reduce the error.
- The network's output is compared to the correct result, we get the error (loss), and then we move backwards updating weights to minimize this error.

Vanishing Gradient is:

- An issue that occurs in deep networks (many layers), especially with certain activation functions like the sigmoid.
- Because the gradient is multiplied many times as it goes back through layers, it can become extremely small, leading the earliest layers to stop learning effectively.

Backpropagation:

- Is the algorithm that makes deep learning feasible. Without it, it would be extremely difficult to train multi-layer networks.
- Uses gradient descent (or variants) to iteratively update weights.
- The gradient of the loss function w.r.t. each weight is computed using partial derivatives. This leverages the chain rule.

Vanishing Gradient:

- Happens when partial derivatives of activation functions become too small.
- Example: Sigmoid activation $\sigma(x) = \frac{1}{1+e^{-x}}$. The derivative is $\sigma'(x) = \sigma(x)(1 - \sigma(x))$. For large positive or negative x, the slope is nearly 0.
- Repeated multiplication of small derivatives across many layers yields a near-zero gradient, stalling training.
- Modern techniques to mitigate it: ReLU activation, batch normalization, careful initialization, skip connections (ResNets), etc.

- Derives from the **reverse-mode differentiation** technique.

- If we define a loss function L and a chain of transformations (layers) $f^{(1)}, f^{(2)}, \ldots, f^{(n)}$, the backprop step is essentially computing $\frac{\partial L}{\partial w_i}$ for each parameter w_i efficiently.
- Complexity is roughly linear in the number of parameters (rather than exponential if you tried naive forward differences).

Vanishing Gradient:

- A major challenge in training Recurrent Neural Networks (RNNs) and older feedforward models with many layers.
- Often tackled by using gating mechanisms (e.g. LSTM, GRU in RNNs) or architectures designed to have shortcuts.
- When the gradient is too small, it doesn't effectively update earlier weights. In extreme cases, those weights become "frozen."

- **Historical context**: Backprop was popularized in the 1980s by Rumelhart, Hinton, and Williams. The concept dates back earlier in control theory and was re-discovered for neural networks. It laid the foundation for modern deep learning.
- **Mathematical foundation**: Based on applying the chain rule to compute gradients of parameterized compositions of functions. Reverse-mode auto-differentiation offers a computationally efficient solution.
- **Vanishing and Exploding Gradients**: Exploding gradients also appear in deep networks. Techniques like gradient clipping, careful initialization, orthogonal or identity-based initialization in RNNs, etc. are used. In deep feed-forward networks, skip connections (e.g., ResNet) help preserve gradient flow.

2. Brief History and Underlying Technology

- Backpropagation was described in a famous 1986 paper by **Rumelhart, Hinton, and Williams**: "Learning representations by back-propagating errors".
- The core idea, *gradient-based optimization*, was well-known in mathematics, but its application to multi-layer neural networks revolutionized the field.
- Early networks faced challenges like computational limits and lack of big datasets. In the mid-2000s, with GPUs and large datasets, backprop-based deep neural networks resurged.

3. Intuition Behind Backpropagation

- You have a neural network making predictions y_{pred} for a target y_{true}.
- Compute the **loss** $L(y_{pred}, y_{true})$. Often Mean Squared Error (MSE) or Cross-Entropy Loss.
- We want to adjust each weight and bias so the loss is minimized.
- We measure **how much** each weight contributed to the error. The chain rule helps us compute these contributions.
- Then we tweak the weights in the direction that reduces the error (this is the essence of gradient descent).

4. Math of Backprop

4.1 Single Neuron Example

- Suppose we have a neuron with input x, weight w, bias b, and output $\hat{y}$.
- The neuron's output: $\hat{y} = \sigma(wx + b)$, where σ is an activation function (e.g. sigmoid).
- Loss function example: $L = \frac{1}{2}(y_{true} - \hat{y})^2$ (MSE).
- We want $\frac{\partial L}{\partial w}$ and $\frac{\partial L}{\partial b}$.

Step by step:

1. $\frac{\partial L}{\partial \hat{y}} = \hat{y} - y_{true}$ because of the derivative of $\frac{1}{2}(error)^2$ w.r.t. $error$.
2. $\frac{\partial \hat{y}}{\partial z} = \sigma'(z)$ where $z = wx + b$.
3. $\frac{\partial z}{\partial w} = x$ and $\frac{\partial z}{\partial b} = 1$.

Chain rule:

- $\frac{\partial L}{\partial w} = \frac{\partial L}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial z} \times \frac{\partial z}{\partial w}$
- $\frac{\partial L}{\partial b} = \frac{\partial L}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial z} \times \frac{\partial z}{\partial b}$

5. Illustrative Example with Code

We'll build a simple 2-layer neural network to illustrate backprop.

5.1 Example Calculations

Assume we have inputs x_1, x_2 and a single hidden layer with weights $W^{(1)}$ and output layer with weights $W^{(2)}$. We'll do a forward pass, compute the loss, then show how partial derivatives are computed. We'll keep it symbolic here. In code, we'll do the actual numeric steps.

- **Weights**: $w_1, w_2, \ldots$
- **Biases**: $b_1, b_2, \ldots$
- We compute the forward pass: $\mathbf{h} = \sigma(W^{(1)}x + b^{(1)})$ -> $\hat{y} = \sigma(W^{(2)}\mathbf{h} + b^{(2)})$
- Then the loss $L = f(\hat{y}, y_{true})$.
- Using the chain rule, we find partial derivatives of L wrt each parameter. Then we do gradient descent: $\theta \leftarrow \theta - \alpha \frac{\partial L}{\partial \theta}$, where α is the learning rate.

5.2 Step by Step Example (From Scratch)

1. **Initialize** random weights: $W^{(1)}$, $b^{(1)}$, $W^{(2)}$, $b^{(2)}$.
2. **Forward pass**: compute hidden layer, then output, then the loss.
3. **Backward pass**: compute partial derivatives of loss wrt each weight and bias.
4. **Update** the weights using gradient descent.
5. **Repeat** until convergence.

6. Illustrative Problem Solved

A simple example is to classify points as either inside or outside a circle (a basic classification). By adjusting the network's weights via backprop, we can teach the neural network to recognize circular boundaries.

7. Real-World Problem

Backprop is used everywhere in deep learning, for example:

- **Image classification** (e.g. is this a cat or a dog?)
- **Speech recognition** (transcribing speech to text)
- **Natural language processing** (LLMs, text generation, translations) In all these tasks, the same principle applies: compute the error, backpropagate, adjust weights.

8. How to Solve a Real-World Problem Using This Tech

1. Collect data (inputs and correct outputs).
2. Design a neural network architecture.
3. Initialize weights.
4. Compute forward pass -> compute loss.
5. Backprop and update weights.
6. Repeat until loss is minimal or acceptable.
7. Use the trained network for predictions on new data.

9. Questions to Ponder

1. Why do we need an activation function?
2. How does the choice of activation function affect vanishing gradients?
3. How does learning rate influence training?
4. How do we mitigate vanishing gradient problems?
5. What are alternative optimizers to vanilla gradient descent?

Answers

1. **Need for activation**: Without it, the network would be just a linear mapping, no matter how many layers. Activations add non-linearity.
2. **Effect on vanishing gradients**: Sigmoid or tanh saturate for large inputs, making gradients small. ReLU avoids saturation in positive region.
3. **Learning rate**: Too high -> might overshoot minima. Too low -> very slow convergence. Must pick carefully.
4. **Mitigating vanishing gradients**: Use ReLU, batch normalization, skip connections, better initializations.
5. **Alternative optimizers**: Momentum, RMSProp, Adam, etc. They can handle varying learning rates, accelerate convergence.

10. Code Examples

10.1 Minimal Working Example (All Completed)

Below is a simple neural network from scratch in Python (using NumPy) that learns a small function.

```
# Let's build and train a simple 2-layer neural network to learn XOR, for example.
# We'll show how backprop is implemented manually.

import numpy as np

# XOR data
X = np.array([[0,0], [0,1], [1,0], [1,1]])
y = np.array([[0], [1], [1], [0]])

np.random.seed(42)
# Initialize weights for layer1 (2 -> 2), layer2 (2 -> 1)
W1 = np.random.randn(2, 2)
b1 = np.zeros((1, 2))
W2 = np.random.randn(2, 1)
b2 = np.zeros((1, 1))

def sigmoid(x):
    return 1.0 / (1.0 + np.exp(-x))

def sigmoid_deriv(x):
    return sigmoid(x) * (1 - sigmoid(x))

lr = 0.1  # learning rate
epochs = 10000

for i in range(epochs):
    # Forward pass
    z1 = np.dot(X, W1) + b1
    a1 = sigmoid(z1)
    z2 = np.dot(a1, W2) + b2
    a2 = sigmoid(z2)

    # Compute loss (mean squared error)
    loss = np.mean((y - a2) ** 2)

    # Backprop
    # dL/da2
    dL_da2 = 2*(a2 - y) / y.shape[0]
    # dL/dz2 = dL/da2 * da2/dz2
    dL_dz2 = dL_da2 * sigmoid_deriv(z2)

    # dL/dW2 = a1^T * dL_dz2
    dW2 = np.dot(a1.T, dL_dz2)
    db2 = np.sum(dL_dz2, axis=0, keepdims=True)

    # dL/dz1 = dL/da1 * da1/dz1
    # but dL/da1 = dL_dz2 * W2^T
    dL_da1 = np.dot(dL_dz2, W2.T)
    dL_dz1 = dL_da1 * sigmoid_deriv(z1)

    dW1 = np.dot(X.T, dL_dz1)
    db1 = np.sum(dL_dz1, axis=0, keepdims=True)

    # Update parameters
    W2 -= lr * dW2
    b2 -= lr * db2
    W1 -= lr * dW1
    b1 -= lr * db1

    if i % 2000 == 0:
        print(f"Epoch {i}, Loss: {loss:.4f}")

print("Final predictions:")
print(a2.round())
```

```
Epoch 0, Loss: 0.2558
Epoch 2000, Loss: 0.2494
Epoch 4000, Loss: 0.2454
Epoch 6000, Loss: 0.2046
Epoch 8000, Loss: 0.1532
Final predictions:
[[0.]
 [0.]
 [1.]
 [1.]]
```

10.2 A Sample Exercise

Below is a skeleton code for you to fill in. We'll compute a similar network for a simple function. Follow the `TODO` comments to complete the missing lines.

```
# TODO: Complete the code and run!
import numpy as np

# Suppose we want to learn the AND function
X = np.array([[0,0], [0,1], [1,0], [1,1]])
y = np.array([[0], [0], [0], [1]])

np.random.seed(0)
W1 = np.random.randn(2, 2)
b1 = np.zeros((1, 2))
W2 = np.random.randn(2, 1)
b2 = np.zeros((1, 1))

def sigmoid(x):
  return 1/(1+np.exp(-x))

def sigmoid_deriv(x):
  # TODO: implement derivative of sigmoid
  return sigmoid(x) * (1 - sigmoid(x))

lr = 0.1
epochs = 5000

for i in range(epochs):
  # Forward pass
  # TODO: compute z1, a1, z2, a2 using W1, b1, W2, b2
  z1 = np.dot(X, W1) + b1
  a1 = sigmoid(z1)
  z2 = np.dot(a1, W2) + b2
  a2 = sigmoid(z2)

  # Loss
  loss = np.mean((y - a2)**2)

  # Backprop
  # TODO: fill in the partial derivatives and parameter updates
  dL_da2 = 2*(a2 - y) / y.shape[0]
  dL_dz2 = dL_da2 * sigmoid_deriv(z2)
  dW2 = np.dot(a1.T, dL_dz2)
  db2 = np.sum(dL_dz2, axis=0, keepdims=True)

  dL_da1 = np.dot(dL_dz2, W2.T)
  dL_dz1 = dL_da1 * sigmoid_deriv(z1)
  dW1 = np.dot(X.T, dL_dz1)
  db1 = np.sum(dL_dz1, axis=0, keepdims=True)

  # Update
  # TODO: update W2, b2, W1, b1 with lr * dW, dB
  W2 -= lr * dW2
  b2 -= lr * db2
  W1 -= lr * dW1
  b1 -= lr * db1

  if i % 1000 == 0:
    print(f"Epoch {i}, Loss: {loss}")

print("Trained!")
print("Test predictions:", a2)
```

```
Epoch 0, Loss: 0.34477440568223433
Epoch 1000, Loss: 0.09469907283773513
Epoch 2000, Loss: 0.034526774761455825
Epoch 3000, Loss: 0.016221608911122266
Epoch 4000, Loss: 0.009673113121277846
Trained!
Test predictions: [[0.02746471]
 [0.08113197]
 [0.08197225]
 [0.88825806]]
```

11. Glossary

- **Backpropagation**: Algorithm for computing gradients of a neural network's parameters in reverse order of layers.

- **Vanishing Gradients**: Situation where gradients become very small in early layers, hindering learning.
- **Chain Rule**: A fundamental calculus rule used to compute the derivative of composite functions.
- **Weight, Bias**: Learnable parameters of a neuron/network. The network adjusts these via training.
- **Activation Function**: A non-linear function (sigmoid, ReLU, etc.) applied at each neuron.
- **Loss Function**: Measures the difference between predictions and ground truth.
- **Gradient**: The direction of steepest ascent for a function. We often move in the negative gradient direction to minimize loss.
- **Learning Rate**: Hyperparameter that controls how big a step to take when updating weights.
- **MSE (Mean Squared Error)**: A common loss function, average of the squares of errors.
- **Cross-Entropy**: Another common loss function, especially for classification.

End of Notebook

```
import os, sys, platform, datetime, uuid, socket

def signoff(name="Anonymous"):
    colab_check = "Yes" if 'google.colab' in sys.modules else "No"
    mac_addr = ':'.join(format((uuid.getnode() >> i) & 0xff, '02x') for i in reversed(range(0, 48, 8)))
    print("+++ Acknowledgement +++")
    print(f"Executed on: {datetime.datetime.now()}")
    print(f"In Google Colab: {colab_check}")
    print(f"System info: {platform.system()} {platform.release()}")
    print(f"Node name: {platform.node()}")
    print(f"MAC address: {mac_addr}")
    try:
        print(f"IP address: {socket.gethostbyname(socket.gethostname())}")
    except:
        print("IP address: Unknown")
    print(f"Signing off, {name}")

signoff("Ali Muhammad Asad")
```

```
+++ Acknowledgement +++
Executed on: 2025-01-28 18:02:12.673953
In Google Colab: No
System info: Linux 6.8.0-51-generic
Node name: alimuhammad-Inspiron-7559
MAC address: 20:47:47:74:94:05
IP address: 127.0.1.1
Signing off, Ali Muhammad Asad
```