Neural Networks

Muskula Rahul

1 Structure of a Neural Network

A neural network is composed of interconnected layers of nodes (neurons), mimicking the human brain. These layers are organized to process data through a series of transformations, allowing the network to learn complex patterns and make predictions. The primary layer types are:

- **Input Layer**: This layer acts as the entry point for the raw data. Each node in the input layer represents a feature or attribute of the input data. The number of nodes in this layer corresponds to the dimensionality of the input.
- **Hidden Layers**: These intermediate layers perform the core computations. Each hidden layer consists of multiple neurons that apply weighted sums and activation functions to the input they receive. Deep networks have multiple hidden layers, enabling them to learn hierarchical representations of the data. The more hidden layers, the more complex the functions the network can approximate.
- **Output Layer**: The final layer produces the network's prediction. The number of output nodes depends on the nature of the task. For example, a binary classification task would have one output node, while a multi-class classification task would have one node for each class.

Within each layer, neurons are interconnected. A **fully connected** or **dense** layer implies that each neuron in a layer is connected to every neuron in the subsequent layer. This interconnectedness facilitates the flow of information and the learning of relationships between features.

The mathematical representation of a neural network's output y is given by:

$$y = f(\mathbf{X}; \mathbf{W}, \mathbf{b})$$

Where:

- $\mathbf{X}$ represents the input vector.
- $\mathbf{W}$ denotes the set of weights associated with the connections between neurons across all layers. These weights are adjusted during training to minimize the error.
- $\mathbf{b}$ represents the set of biases for each neuron across all layers. Biases introduce an offset, allowing the network to model more complex functions.
- f symbolizes the overall function computed by the network. This function is typically a composition of linear transformations (weighted sums plus biases) and non-linear activation functions.

2 Forward Propagation

Forward propagation is the process of passing input data through the neural network to generate an output prediction. This process involves a series of calculations at each layer, starting from the input layer and proceeding sequentially to the output layer.

2.1 Linear Transformation

Each neuron in a layer performs a linear transformation on its inputs. This involves calculating a weighted sum of the inputs and adding a bias term. Mathematically, the output $z_i^{(l)}$ of neuron i in layer l is expressed as:

$$z_i^{(l)} = \sum_j w_{ij}^{(l)} a_j^{(l-1)} + b_i^{(l)}$$

where:

- $w_{ij}^{(l)}$ is the weight connecting neuron j in layer $l-1$ to neuron i in layer l.
- $a_j^{(l-1)}$ is the activation (output) of neuron j in the previous layer $(l-1)$. For the input layer, $a^{(0)}$ is the input vector $\mathbf{X}$.
- $b_i^{(l)}$ is the bias term associated with neuron i in layer l.

2.2 Activation Functions

Activation functions introduce non-linearity into the network. This non-linearity is essential for enabling the network to model complex, non-linear relationships in the data. Without activation functions, the network would simply be a series of linear transformations, limiting its expressive power. Common activation functions include:

- **Sigmoid**: $\sigma(z) = \frac{1}{1+e^{-z}}$ — Outputs values between 0 and 1, often used in the output layer for binary classification.
- **ReLU (Rectified Linear Unit)**: $\text{ReLU}(z) = \max(0, z)$ — Computationally efficient and effective in mitigating the vanishing gradient problem. Popular in hidden layers.
- **Leaky ReLU**: $\text{Leaky ReLU}(z) = \begin{cases} z, & z > 0 \\ 0.01z, & z \leq 0 \end{cases}$ — Addresses the "dying ReLU" issue by allowing a small gradient for negative inputs.
- **Tanh (Hyperbolic Tangent)**: $\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$ — Outputs values between -1 and 1, sometimes preferred over sigmoid.

The activation function is applied to the linear transformation output (z), resulting in the activation $a_i^{(l)}$: $a_i^{(l)} = g(z_i^{(l)})$ where g is the activation function.

2.3 Forward Propagation Example

```python
import numpy as np

def relu(z):
    return np.maximum(0, z)

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

def forward_propagation(X, layers, activations):
    cache = {} # Store intermediate values for backpropagation
    a = X
    for i, (weights, biases) in enumerate(layers):
        z = np.dot(a, weights) + biases
        a = activations[i](z)
        cache[f"z{i+1}"] = z
        cache[f"a{i+1}"] = a
    return a, cache # Return final activation and cached values
```

In this code:

- `layers`: A list of tuples, where each tuple contains the weight matrix and bias vector for a layer.
- `activations`: A list of activation functions, one for each layer.
- `cache`: Stores intermediate values (z and a) for each layer, which are needed during backpropagation.

3 Loss Functions

A loss function quantifies the error between the network's predictions and the true target values. The choice of loss function depends on the specific task:

3.1 Classification Loss Functions

Binary Cross-Entropy: Used for binary classification problems (e.g., spam/not spam). It measures the dissimilarity between the predicted probability distribution and the true distribution.

$$L = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)\right]$$

where N is the number of samples, y_i is the true label (0 or 1), and $\hat{y}_i$ is the predicted probability.

Categorical Cross-Entropy: Used for multi-class classification problems (e.g., image classification). It generalizes binary cross-entropy to multiple classes.

$$L = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y_{ic}\log(\hat{y}_{ic})$$

where C is the number of classes, y_{ic} is 1 if sample i belongs to class c (and 0 otherwise), and $\hat{y}_{ic}$ is the predicted probability for class c.

3.2 Regression Loss Functions

Mean Squared Error (MSE): Commonly used for regression tasks (e.g., predicting house prices). It calculates the average squared difference between predicted and true values.

$$L = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

Mean Absolute Error (MAE): Another regression loss function that measures the average absolute difference between predicted and true values. Less sensitive to outliers than MSE.

$$L = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i|$$

The goal of training is to minimize the chosen loss function, bringing the network's predictions closer to the true values.

4 Backward Propagation

Backward propagation, or backpropagation, is the central algorithm for training neural networks. It efficiently calculates the gradients of the loss function with respect to the network's weights and biases. These gradients are then used to update the weights and biases, iteratively improving the network's performance.

4.1 Chain Rule and Gradient Computation

Backpropagation relies on the chain rule of calculus. The chain rule allows us to decompose the gradient of the loss function with respect to a weight w into a product of partial derivatives:

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w}$$

This allows us to calculate the gradient by starting from the output layer and working backward, layer by layer. The gradients are accumulated as the error signal propagates through the network.

4.2 Optimizing with Gradient Descent

Gradient descent uses the calculated gradients to update the weights and biases, moving them in the direction that reduces the loss. The basic update rule is:

$$w := w - \alpha\frac{\partial L}{\partial w}$$

where α is the learning rate, a hyperparameter that controls the step size of the updates.

Advanced Optimization Techniques

- **Momentum**: Adds a fraction of the previous update to the current update, accelerating convergence and helping to escape local minima.
- **Adam (Adaptive Moment Estimation)**: Combines momentum with adaptive learning rates for each parameter, providing efficient and robust optimization.
- **RMSprop (Root Mean Square Propagation)**: Another adaptive learning rate method that addresses the vanishing gradient problem.

4.3 Backpropagation Code Example

```python
def backward_propagation(X, y, weights, biases, cache, learning_rate=0.01):
    # Retrieve cached values
    a = cache["a1"]
    z = cache["z1"]

    # Calculate gradients (example for binary cross-entropy and sigmoid)
    m = X.shape[0] # Number of samples
    dz = a - y  # Gradient of loss w.r.t. z
    dw = (1/m) * np.dot(X.T, dz) # Gradient of loss w.r.t. weights
    db = (1/m) * np.sum(dz) # Gradient of loss w.r.t. bias

    # Update weights and biases
    weights -= learning_rate * dw
    biases -= learning_rate * db

    return weights, biases
```

5 Training the Neural Network

Training a neural network involves repeatedly performing forward propagation, calculating the loss, performing backward propagation to compute gradients, and updating the weights and biases. This iterative process continues for a specified number of epochs or until a desired level of performance is achieved.

Regularization Techniques

Regularization techniques help prevent overfitting, where the network performs well on the training data but poorly on unseen data.

L1 Regularization: Adds a penalty proportional to the absolute value of the weights to the loss function. Encourages sparsity in the weights. The L1 regularization term can be expressed as:

$$L_1 = \lambda \sum_{i=1}^{n} |w_i|$$

where λ is the regularization strength and w_i are the model weights.

L2 Regularization (Weight Decay): Adds a penalty proportional to the square of the weights to the loss function. Helps prevent weights from becoming too large. The L2 regularization term can be written as:

$$L_2 = \frac{\lambda}{2} \sum_{i=1}^{n} w_i^2$$

Dropout: Randomly deactivates neurons during training, forcing the network to learn more robust features.

Training Loop with Regularization and Adam Optimizer

```python
def train(X, y, layers, epochs=100, learning_rate=0.001, beta1=0.9, beta2=0.999,
epsilon=1e-8):
    # Initialize weights, biases, and Adam parameters
    m, v = {}, {}
    for i, (weights, biases) in enumerate(layers):
        m[i], v[i] = np.zeros_like(weights), np.zeros_like(weights)

    for epoch in range(epochs):
        # Forward propagation
        a, cache = forward_propagation(X, layers, activations=[relu]*len(layers))

        # Calculate loss
        loss = -np.mean(y * np.log(a) + (1 - y) * np.log(1 - a))

        # Backward propagation and Adam update
        for i in reversed(range(len(layers))):
            weights, biases = layers[i]
            m[i], v[i] = update_parameters_with_adam(weights, biases, m[i], v[i],
            learning_rate, epoch, beta1, beta2, epsilon)

 if epoch % 10 == 0:
            print(f"Epoch {epoch}, Loss: {loss}")

    return layers
```

Conclusion

Neural networks are powerful tools for learning complex patterns from data. Forward and backward propagation are the core algorithms that drive their training. The choice of architecture, loss function, optimizer, and regularization techniques plays a crucial role in the success of a neural network application. Continuing research and development in the field of deep learning are constantly expanding the capabilities and applications of neural networks.