

**Building Neural Networks and Transformers From Scratch in Python**

This comprehensive guide explores how to implement neural networks and transformer models from scratch in Python, using only fundamental libraries like NumPy. By building these powerful machine learning models without relying on deep learning frameworks, you'll gain an intimate understanding of their inner workings.

**Part I: Creating a Neural Network from Scratch**

**Introduction to Neural Networks**

Neural networks are computational models inspired by the human brain that learn from data through a process of optimization. At their core, they consist of interconnected neurons organized in layers that process information and make predictions[[1]](#fn1).

**Mathematical Foundations**

Before diving into implementation, let's understand the key mathematical concepts:

1. **Neurons and Activation**: A neuron takes multiple inputs, multiplies them by weights, adds a bias term, and applies an activation function to produce an output[[2]](#fn2).
2. **Feed-Forward Propagation**: Information flows from input to output through weighted connections[[2]](#fn2).
3. **Loss Function**: Measures the difference between predicted and actual outputs[[2]](#fn2).
4. **Gradient Descent**: An optimization algorithm that adjusts weights to minimize the loss function[[2]](#fn2).
5. **Backpropagation**: The algorithm that calculates gradients and updates weights in the network[[2]](#fn2).

**Implementing a Simple Neural Network**

Let's implement a basic neural network from scratch using only Python and NumPy. We'll create a multilayer feed-forward network that can be trained using backpropagation[[3]](#fn3)[[2]](#fn2).

**Setting Up**

First, import NumPy for matrix operations:

import numpy as np

**Neural Network Class**

class NeuralNetwork:  
 def \_\_init\_\_(self, layers, activation='sigmoid'):  
 """  
 Initialize neural network with specified layer sizes.  
   
 Parameters:  
 - layers: List of integers, each specifying the number of neurons in a layer  
 - activation: Activation function to use ('sigmoid' or 'relu')  
 """  
 self.layers = layers  
 self.activation = activation  
   
 # Initialize weights and biases  
 self.weights = []  
 self.biases = []  
   
 # Random weights initialization between layers  
 for i in range(len(layers) - 1):  
 # Using Xavier/Glorot initialization for weights  
 w = np.random.randn(layers[i], layers[i+1]) \* np.sqrt(1 / layers[i])  
 b = np.zeros((1, layers[i+1]))  
   
 self.weights.append(w)  
 self.biases.append(b)

**Activation Functions**

We'll implement common activation functions and their derivatives[[2]](#fn2)[[4]](#fn4):

def sigmoid(x):  
 return 1 / (1 + np.exp(-x))  
  
def sigmoid\_derivative(x):  
 return sigmoid(x) \* (1 - sigmoid(x))  
  
def relu(x):  
 return np.maximum(0, x)  
  
def relu\_derivative(x):  
 return np.where(x > 0, 1, 0)

**Forward Propagation**

The forward pass computes network predictions from inputs[[2]](#fn2):

def forward\_propagation(self, X):  
 """  
 Perform forward propagation through the network.  
   
 Parameters:  
 - X: Input data of shape (n\_samples, n\_features)  
   
 Returns:  
 - List of activations for each layer  
 """  
 activations = [X] # List to store all activations  
   
 for i in range(len(self.weights)):  
 # Calculate net input to layer  
 z = np.dot(activations[-1], self.weights[i]) + self.biases[i]  
   
 # Apply activation function  
 if self.activation == 'sigmoid':  
 a = sigmoid(z)  
 elif self.activation == 'relu':  
 a = relu(z)  
   
 activations.append(a)  
   
 return activations

**Backpropagation**

The backpropagation algorithm computes gradients and updates weights[[2]](#fn2)[[1]](#fn1):

def backward\_propagation(self, X, y, learning\_rate=0.1):  
 """  
 Perform backpropagation to update network weights.  
   
 Parameters:  
 - X: Input data  
 - y: Target values  
 - learning\_rate: Learning rate for weight updates  
   
 Returns:  
 - Updated weights and biases  
 """  
 # Forward pass  
 activations = self.forward\_propagation(X)  
   
 # Calculate output layer error  
 output\_error = activations[-1] - y  
   
 # Initialize list to store all errors  
 errors = [output\_error]  
   
 # Backpropagate error through all layers  
 for i in range(len(self.weights) - 1, 0, -1):  
 # Calculate error for current layer  
 if self.activation == 'sigmoid':  
 error = np.dot(errors[^0], self.weights[i].T) \* sigmoid\_derivative(activations[i])  
 elif self.activation == 'relu':  
 error = np.dot(errors[^0], self.weights[i].T) \* relu\_derivative(activations[i])  
   
 # Insert at beginning of list  
 errors.insert(0, error)  
   
 # Update weights and biases using gradient descent  
 for i in range(len(self.weights)):  
 self.weights[i] -= learning\_rate \* np.dot(activations[i].T, errors[i])  
 self.biases[i] -= learning\_rate \* np.sum(errors[i], axis=0, keepdims=True)

**Training Loop**

Now let's create a training function that iteratively improves the network[[2]](#fn2):

def train(self, X, y, epochs=1000, learning\_rate=0.1, verbose=True):  
 """  
 Train the neural network.  
   
 Parameters:  
 - X: Training inputs  
 - y: Target outputs  
 - epochs: Number of training iterations  
 - learning\_rate: Learning rate for weight updates  
 - verbose: Whether to print progress  
 """  
 for epoch in range(epochs):  
 # Forward and backward passes  
 self.backward\_propagation(X, y, learning\_rate)  
   
 # Calculate and print loss if verbose  
 if verbose and epoch % 100 == 0:  
 activations = self.forward\_propagation(X)  
 predictions = activations[-1]  
 loss = np.mean(np.square(predictions - y))  
 print(f"Epoch {epoch}, Loss: {loss:.6f}")  
   
 print("Training complete!")

**Prediction Function**

Finally, a function to make predictions with the trained network[[3]](#fn3):

def predict(self, X):  
 """  
 Generate predictions for inputs.  
   
 Parameters:  
 - X: Input data  
   
 Returns:  
 - Predictions  
 """  
 activations = self.forward\_propagation(X)  
 return activations[-1]

**Example: XOR Problem**

Let's test our neural network on the XOR problem, which requires a non-linear solution[[3]](#fn3)[[4]](#fn4):

# Create XOR dataset  
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])  
y = np.array([[^0], [^1], [^1], [^0]])  
  
# Initialize and train network  
nn = NeuralNetwork([2, 4, 1])  
nn.train(X, y, epochs=10000, learning\_rate=0.1)  
  
# Test predictions  
predictions = nn.predict(X)  
print("Predictions:")  
for i in range(len(X)):  
 print(f"Input: {X[i]} -> Output: {predictions[i][^0]:.4f}, Expected: {y[i][^0]}")

**Part II: Creating a Transformer Algorithm from Scratch**

**Introduction to Transformers**

Transformers are neural network architectures that revolutionized natural language processing by introducing a self-attention mechanism that allows the model to focus on different parts of the input sequence[[5]](#fn5)[[6]](#fn6). Unlike recurrent neural networks, transformers process all elements of a sequence in parallel, making them more efficient to train[[7]](#fn7).

**Core Components of Transformers**

Let's implement each component of the transformer architecture from scratch[[5]](#fn5)[[6]](#fn6)[[7]](#fn7):

**1. Input Embeddings**

The first step is to convert input tokens into continuous vector representations[[8]](#fn8):

class Embeddings(nn.Module):  
 def \_\_init\_\_(self, vocab\_size, embedding\_dim):  
 """  
 Initialize the embedding layer.  
   
 Parameters:  
 - vocab\_size: Size of the vocabulary  
 - embedding\_dim: Dimension of embeddings  
 """  
 super().\_\_init\_\_()  
 self.embedding\_dim = embedding\_dim  
 self.embedding = np.random.randn(vocab\_size, embedding\_dim) \* 0.01  
   
 def forward(self, x):  
 """  
 Convert token indices to embeddings.  
   
 Parameters:  
 - x: Input token indices  
   
 Returns:  
 - Embeddings for each token  
 """  
 return self.embedding[x]

**2. Positional Encoding**

Since transformers don't have a sequential nature, we need to add positional information to the embeddings[[9]](#fn9)[[7]](#fn7):

def get\_positional\_encoding(seq\_length, d\_model, n=10000):  
 """  
 Create positional encoding matrix.  
   
 Parameters:  
 - seq\_length: Maximum sequence length  
 - d\_model: Dimension of the model  
 - n: Scaling factor  
   
 Returns:  
 - Positional encoding matrix  
 """  
 # Initialize encoding matrix  
 position\_enc = np.zeros((seq\_length, d\_model))  
   
 # Calculate positional encodings  
 for pos in range(seq\_length):  
 for i in range(0, d\_model, 2):  
 # Sine for even indices  
 position\_enc[pos, i] = np.sin(pos / (n \*\* (i / d\_model)))  
   
 # Cosine for odd indices  
 if i + 1 < d\_model:  
 position\_enc[pos, i + 1] = np.cos(pos / (n \*\* (i / d\_model)))  
   
 return position\_enc

**3. Self-Attention Mechanism**

The self-attention mechanism allows the model to weigh the importance of different tokens in the sequence[[6]](#fn6)[[7]](#fn7):

def scaled\_dot\_product\_attention(query, key, value, mask=None):  
 """  
 Compute scaled dot-product attention.  
   
 Parameters:  
 - query: Query vectors  
 - key: Key vectors  
 - value: Value vectors  
 - mask: Optional mask for padding or future tokens  
   
 Returns:  
 - Context vectors and attention weights  
 """  
 # Calculate dot product of query and key  
 matmul\_qk = np.dot(query, key.T)  
   
 # Scale dot product  
 d\_k = query.shape[-1]  
 scaled\_attention\_logits = matmul\_qk / np.sqrt(d\_k)  
   
 # Apply mask if provided  
 if mask is not None:  
 scaled\_attention\_logits += (mask \* -1e9)  
   
 # Apply softmax to get attention weights  
 attention\_weights = softmax(scaled\_attention\_logits, axis=-1)  
   
 # Multiply weights by values  
 output = np.dot(attention\_weights, value)  
   
 return output, attention\_weights  
  
def softmax(x, axis=-1):  
 """  
 Compute softmax values for each set of scores.  
   
 Parameters:  
 - x: Input array  
 - axis: Axis along which to compute softmax  
   
 Returns:  
 - Softmax values  
 """  
 exp\_x = np.exp(x - np.max(x, axis=axis, keepdims=True))  
 return exp\_x / np.sum(exp\_x, axis=axis, keepdims=True)

**4. Multi-Head Attention**

Multi-head attention allows the model to attend to information from different representation subspaces[[6]](#fn6)[[7]](#fn7):

class MultiHeadAttention:  
 def \_\_init\_\_(self, d\_model, num\_heads):  
 """  
 Initialize multi-head attention layer.  
   
 Parameters:  
 - d\_model: Dimension of the model  
 - num\_heads: Number of attention heads  
 """  
 assert d\_model % num\_heads == 0, "d\_model must be divisible by num\_heads"  
   
 self.d\_model = d\_model  
 self.num\_heads = num\_heads  
 self.depth = d\_model // num\_heads  
   
 # Define weight matrices  
 self.wq = np.random.randn(d\_model, d\_model) \* 0.01  
 self.wk = np.random.randn(d\_model, d\_model) \* 0.01  
 self.wv = np.random.randn(d\_model, d\_model) \* 0.01  
 self.wo = np.random.randn(d\_model, d\_model) \* 0.01  
   
 def split\_heads(self, x, batch\_size):  
 """  
 Split the last dimension into (num\_heads, depth).  
   
 Parameters:  
 - x: Input tensor  
 - batch\_size: Batch size  
   
 Returns:  
 - Reshaped tensor  
 """  
 x = x.reshape(batch\_size, -1, self.num\_heads, self.depth)  
 return x.transpose(0, 2, 1, 3)  
   
 def forward(self, query, key, value, mask=None):  
 """  
 Compute multi-head attention.  
   
 Parameters:  
 - query: Query vectors  
 - key: Key vectors  
 - value: Value vectors  
 - mask: Optional mask  
   
 Returns:  
 - Output after multi-head attention  
 """  
 batch\_size = query.shape[^0]  
   
 # Linear projections  
 q = np.dot(query, self.wq) # (batch\_size, seq\_len, d\_model)  
 k = np.dot(key, self.wk) # (batch\_size, seq\_len, d\_model)  
 v = np.dot(value, self.wv) # (batch\_size, seq\_len, d\_model)  
   
 # Split heads  
 q = self.split\_heads(q, batch\_size) # (batch\_size, num\_heads, seq\_len, depth)  
 k = self.split\_heads(k, batch\_size) # (batch\_size, num\_heads, seq\_len, depth)  
 v = self.split\_heads(v, batch\_size) # (batch\_size, num\_heads, seq\_len, depth)  
   
 # Scaled dot-product attention for each head  
 scaled\_attention = np.zeros\_like(q)  
 attention\_weights = np.zeros((batch\_size, self.num\_heads, q.shape[^2], k.shape[^2]))  
   
 for i in range(batch\_size):  
 for h in range(self.num\_heads):  
 scaled\_attention[i, h], attention\_weights[i, h] = scaled\_dot\_product\_attention(  
 q[i, h], k[i, h], v[i, h], mask  
 )  
   
 # Reshape output  
 scaled\_attention = scaled\_attention.transpose(0, 2, 1, 3) # (batch\_size, seq\_len, num\_heads, depth)  
 concat\_attention = scaled\_attention.reshape(batch\_size, -1, self.d\_model) # (batch\_size, seq\_len, d\_model)  
   
 # Final linear projection  
 output = np.dot(concat\_attention, self.wo) # (batch\_size, seq\_len, d\_model)  
   
 return output, attention\_weights

**5. Feed-Forward Network**

Each transformer block contains a feed-forward network with two linear transformations and a ReLU activation[[6]](#fn6)[[7]](#fn7):

class PositionwiseFeedForward:  
 def \_\_init\_\_(self, d\_model, d\_ff):  
 """  
 Initialize feed-forward network.  
   
 Parameters:  
 - d\_model: Dimension of the model  
 - d\_ff: Dimension of the feed-forward network  
 """  
 self.w1 = np.random.randn(d\_model, d\_ff) \* 0.01  
 self.b1 = np.zeros(d\_ff)  
 self.w2 = np.random.randn(d\_ff, d\_model) \* 0.01  
 self.b2 = np.zeros(d\_model)  
   
 def forward(self, x):  
 """  
 Compute feed-forward network output.  
   
 Parameters:  
 - x: Input tensor  
   
 Returns:  
 - Output after feed-forward network  
 """  
 # First linear transformation with ReLU  
 hidden = np.maximum(0, np.dot(x, self.w1) + self.b1)  
   
 # Second linear transformation  
 output = np.dot(hidden, self.w2) + self.b2  
   
 return output

**6. Layer Normalization**

Layer normalization stabilizes the learning process of deep networks[[7]](#fn7)[[8]](#fn8):

def layer\_norm(x, epsilon=1e-6):  
 """  
 Apply layer normalization.  
   
 Parameters:  
 - x: Input tensor  
 - epsilon: Small constant for numerical stability  
   
 Returns:  
 - Normalized tensor  
 """  
 mean = np.mean(x, axis=-1, keepdims=True)  
 variance = np.var(x, axis=-1, keepdims=True)  
   
 normalized = (x - mean) / np.sqrt(variance + epsilon)  
   
 # Note: In a complete implementation, we would also include   
 # learnable scale and shift parameters  
   
 return normalized

**7. Encoder Block**

The encoder block combines multi-head attention, feed-forward networks, and residual connections[[7]](#fn7)[[5]](#fn5):

class EncoderBlock:  
 def \_\_init\_\_(self, d\_model, num\_heads, d\_ff, dropout\_rate=0.1):  
 """  
 Initialize encoder block.  
   
 Parameters:  
 - d\_model: Dimension of the model  
 - num\_heads: Number of attention heads  
 - d\_ff: Dimension of the feed-forward network  
 - dropout\_rate: Dropout rate  
 """  
 self.mha = MultiHeadAttention(d\_model, num\_heads)  
 self.ffn = PositionwiseFeedForward(d\_model, d\_ff)  
 self.dropout\_rate = dropout\_rate  
   
 def forward(self, x, mask=None):  
 """  
 Compute encoder block output.  
   
 Parameters:  
 - x: Input tensor  
 - mask: Optional mask  
   
 Returns:  
 - Output after encoder block  
 """  
 # Multi-head attention with residual connection and layer normalization  
 attn\_output, \_ = self.mha.forward(x, x, x, mask)  
 attn\_output = self.apply\_dropout(attn\_output)  
 out1 = layer\_norm(x + attn\_output)  
   
 # Feed-forward network with residual connection and layer normalization  
 ffn\_output = self.ffn.forward(out1)  
 ffn\_output = self.apply\_dropout(ffn\_output)  
 out2 = layer\_norm(out1 + ffn\_output)  
   
 return out2  
   
 def apply\_dropout(self, x):  
 """  
 Apply dropout to input tensor.  
   
 Parameters:  
 - x: Input tensor  
   
 Returns:  
 - Output after dropout  
 """  
 if self.dropout\_rate > 0:  
 mask = np.random.binomial(1, 1 - self.dropout\_rate, size=x.shape) / (1 - self.dropout\_rate)  
 return x \* mask  
 return x

**8. Decoder Block**

The decoder block is similar to the encoder but adds masked multi-head attention[[7]](#fn7)[[6]](#fn6):

class DecoderBlock:  
 def \_\_init\_\_(self, d\_model, num\_heads, d\_ff, dropout\_rate=0.1):  
 """  
 Initialize decoder block.  
   
 Parameters:  
 - d\_model: Dimension of the model  
 - num\_heads: Number of attention heads  
 - d\_ff: Dimension of the feed-forward network  
 - dropout\_rate: Dropout rate  
 """  
 self.mha1 = MultiHeadAttention(d\_model, num\_heads)  
 self.mha2 = MultiHeadAttention(d\_model, num\_heads)  
 self.ffn = PositionwiseFeedForward(d\_model, d\_ff)  
 self.dropout\_rate = dropout\_rate  
   
 def forward(self, x, enc\_output, look\_ahead\_mask=None, padding\_mask=None):  
 """  
 Compute decoder block output.  
   
 Parameters:  
 - x: Input tensor  
 - enc\_output: Output from encoder  
 - look\_ahead\_mask: Mask for future tokens  
 - padding\_mask: Mask for padding tokens  
   
 Returns:  
 - Output after decoder block  
 """  
 # Masked multi-head attention with residual connection and layer normalization  
 attn1\_output, \_ = self.mha1.forward(x, x, x, look\_ahead\_mask)  
 attn1\_output = self.apply\_dropout(attn1\_output)  
 out1 = layer\_norm(x + attn1\_output)  
   
 # Multi-head attention with encoder output, residual connection, and layer normalization  
 attn2\_output, \_ = self.mha2.forward(out1, enc\_output, enc\_output, padding\_mask)  
 attn2\_output = self.apply\_dropout(attn2\_output)  
 out2 = layer\_norm(out1 + attn2\_output)  
   
 # Feed-forward network with residual connection and layer normalization  
 ffn\_output = self.ffn.forward(out2)  
 ffn\_output = self.apply\_dropout(ffn\_output)  
 out3 = layer\_norm(out2 + ffn\_output)  
   
 return out3  
   
 def apply\_dropout(self, x):  
 """Apply dropout to input tensor."""  
 if self.dropout\_rate > 0:  
 mask = np.random.binomial(1, 1 - self.dropout\_rate, size=x.shape) / (1 - self.dropout\_rate)  
 return x \* mask  
 return x

**9. Full Transformer Architecture**

Now we can assemble the complete transformer model[[5]](#fn5)[[6]](#fn6)[[7]](#fn7):

class Transformer:  
 def \_\_init\_\_(self, vocab\_size, d\_model, num\_heads, d\_ff, num\_layers, max\_seq\_length, dropout\_rate=0.1):  
 """  
 Initialize transformer model.  
   
 Parameters:  
 - vocab\_size: Size of the vocabulary  
 - d\_model: Dimension of the model  
 - num\_heads: Number of attention heads  
 - d\_ff: Dimension of the feed-forward network  
 - num\_layers: Number of encoder and decoder layers  
 - max\_seq\_length: Maximum sequence length  
 - dropout\_rate: Dropout rate  
 """  
 self.d\_model = d\_model  
 self.embedding = Embeddings(vocab\_size, d\_model)  
 self.pos\_encoding = get\_positional\_encoding(max\_seq\_length, d\_model)  
   
 # Encoder and decoder layers  
 self.encoder\_layers = [EncoderBlock(d\_model, num\_heads, d\_ff, dropout\_rate) for \_ in range(num\_layers)]  
 self.decoder\_layers = [DecoderBlock(d\_model, num\_heads, d\_ff, dropout\_rate) for \_ in range(num\_layers)]  
   
 # Final linear layer and softmax  
 self.final\_layer = np.random.randn(d\_model, vocab\_size) \* 0.01  
 self.dropout\_rate = dropout\_rate  
   
 def encode(self, x, mask=None):  
 """  
 Encode input sequence.  
   
 Parameters:  
 - x: Input token indices  
 - mask: Optional mask  
   
 Returns:  
 - Encoded representation  
 """  
 # Input embedding and positional encoding  
 seq\_length = x.shape[^1]  
 x = self.embedding.forward(x)  
 x = x + self.pos\_encoding[:seq\_length, :]  
 x = self.apply\_dropout(x)  
   
 # Pass through encoder layers  
 for layer in self.encoder\_layers:  
 x = layer.forward(x, mask)  
   
 return x  
   
 def decode(self, x, enc\_output, look\_ahead\_mask=None, padding\_mask=None):  
 """  
 Decode input sequence with encoder output.  
   
 Parameters:  
 - x: Input token indices  
 - enc\_output: Output from encoder  
 - look\_ahead\_mask: Mask for future tokens  
 - padding\_mask: Mask for padding tokens  
   
 Returns:  
 - Decoded representation  
 """  
 # Input embedding and positional encoding  
 seq\_length = x.shape[^1]  
 x = self.embedding.forward(x)  
 x = x + self.pos\_encoding[:seq\_length, :]  
 x = self.apply\_dropout(x)  
   
 # Pass through decoder layers  
 for layer in self.decoder\_layers:  
 x = layer.forward(x, enc\_output, look\_ahead\_mask, padding\_mask)  
   
 return x  
   
 def forward(self, inp, tar, enc\_padding\_mask=None, look\_ahead\_mask=None, dec\_padding\_mask=None):  
 """  
 Compute transformer output.  
   
 Parameters:  
 - inp: Input sequence  
 - tar: Target sequence  
 - enc\_padding\_mask: Mask for encoder padding  
 - look\_ahead\_mask: Mask for decoder future tokens  
 - dec\_padding\_mask: Mask for decoder padding  
   
 Returns:  
 - Final output probabilities  
 """  
 # Encode input  
 enc\_output = self.encode(inp, enc\_padding\_mask)  
   
 # Decode with encoder output  
 dec\_output = self.decode(tar, enc\_output, look\_ahead\_mask, dec\_padding\_mask)  
   
 # Final linear layer  
 output = np.dot(dec\_output, self.final\_layer)  
   
 # Apply softmax  
 final\_output = np.zeros\_like(output)  
 for i in range(output.shape[^0]):  
 for j in range(output.shape[^1]):  
 final\_output[i, j] = softmax(output[i, j])  
   
 return final\_output  
   
 def apply\_dropout(self, x):  
 """Apply dropout to input tensor."""  
 if self.dropout\_rate > 0:  
 mask = np.random.binomial(1, 1 - self.dropout\_rate, size=x.shape) / (1 - self.dropout\_rate)  
 return x \* mask  
 return x

**Part III: Pre-training and Testing the Model**

**Data Preparation**

Before pre-training, we need to prepare our dataset[[10]](#fn10)[[11]](#fn11):

def prepare\_data(text\_corpus, vocab\_size, max\_seq\_length):  
 """  
 Prepare data for pre-training.  
   
 Parameters:  
 - text\_corpus: Raw text corpus  
 - vocab\_size: Size of vocabulary  
 - max\_seq\_length: Maximum sequence length  
   
 Returns:  
 - Tokenized sequences and vocabulary  
 """  
 # Create simple tokenizer (character-level for simplicity)  
 chars = sorted(list(set(text\_corpus)))  
 char\_to\_idx = {ch: i for i, ch in enumerate(chars)}  
 idx\_to\_char = {i: ch for i, ch in enumerate(chars)}  
   
 # Tokenize corpus  
 tokenized = [char\_to\_idx[ch] for ch in text\_corpus]  
   
 # Create sequences  
 sequences = []  
 for i in range(0, len(tokenized) - max\_seq\_length, max\_seq\_length):  
 seq = tokenized[i:i + max\_seq\_length]  
 if len(seq) == max\_seq\_length:  
 input\_seq = seq[:-1]  
 target\_seq = seq[1:]  
 sequences.append((input\_seq, target\_seq))  
   
 return sequences, char\_to\_idx, idx\_to\_char

**Pre-training the Model**

To pre-train the model, we'll use a simple language modeling task where the model predicts the next token in a sequence[[10]](#fn10)[[11]](#fn11):

def pre\_train(model, sequences, epochs=10, batch\_size=32, learning\_rate=0.001):  
 """  
 Pre-train the transformer model.  
   
 Parameters:  
 - model: Transformer model  
 - sequences: Training sequences  
 - epochs: Number of training epochs  
 - batch\_size: Batch size  
 - learning\_rate: Learning rate  
   
 Returns:  
 - Trained model and training losses  
 """  
 losses = []  
   
 for epoch in range(epochs):  
 epoch\_loss = 0  
 np.random.shuffle(sequences)  
   
 for i in range(0, len(sequences), batch\_size):  
 batch\_sequences = sequences[i:i + batch\_size]  
   
 # Prepare input and target batches  
 inputs = np.array([seq[^0] for seq in batch\_sequences])  
 targets = np.array([seq[^1] for seq in batch\_sequences])  
   
 # Forward pass  
 # Create necessary masks (simplified)  
 look\_ahead\_mask = create\_look\_ahead\_mask(inputs.shape[^1])  
 output = model.forward(inputs, targets[:, :-1], look\_ahead\_mask=look\_ahead\_mask)  
   
 # Calculate loss and gradients  
 # Note: In a complete implementation, we would perform proper backpropagation here  
 # This is a placeholder for the actual training step  
 batch\_loss = calculate\_loss(output, targets)  
 epoch\_loss += batch\_loss  
   
 # Update weights  
 # This is where we would apply gradient descent updates  
   
 avg\_epoch\_loss = epoch\_loss / (len(sequences) / batch\_size)  
 losses.append(avg\_epoch\_loss)  
   
 print(f"Epoch {epoch+1}/{epochs}, Loss: {avg\_epoch\_loss:.4f}")  
   
 return model, losses  
  
def create\_look\_ahead\_mask(size):  
 """  
 Create look-ahead mask for decoder self-attention.  
   
 Parameters:  
 - size: Size of the sequence  
   
 Returns:  
 - Look-ahead mask  
 """  
 mask = 1 - np.triu(np.ones((size, size)), k=1)  
 return mask  
  
def calculate\_loss(predictions, targets):  
 """  
 Calculate cross-entropy loss.  
   
 Parameters:  
 - predictions: Model predictions  
 - targets: Target values  
   
 Returns:  
 - Loss value  
 """  
 # One-hot encode targets  
 targets\_one\_hot = np.zeros((targets.shape[^0], targets.shape[^1], predictions.shape[-1]))  
 for i in range(targets.shape[^0]):  
 for j in range(targets.shape[^1]):  
 targets\_one\_hot[i, j, targets[i, j]] = 1  
   
 # Calculate cross-entropy loss  
 epsilon = 1e-10  
 loss = -np.sum(targets\_one\_hot \* np.log(predictions + epsilon)) / (targets.shape[^0] \* targets.shape[^1])  
   
 return loss

**Testing the Model**

After pre-training, we evaluate the model's performance[[11]](#fn11):

def test\_model(model, test\_sequences, char\_to\_idx, idx\_to\_char):  
 """  
 Test the transformer model.  
   
 Parameters:  
 - model: Trained transformer model  
 - test\_sequences: Testing sequences  
 - char\_to\_idx: Character to index mapping  
 - idx\_to\_char: Index to character mapping  
   
 Returns:  
 - Test results  
 """  
 correct\_predictions = 0  
 total\_predictions = 0  
   
 for seq in test\_sequences:  
 inputs = np.array([seq[^0]])  
 targets = np.array([seq[^1]])  
   
 # Create masks  
 look\_ahead\_mask = create\_look\_ahead\_mask(inputs.shape[^1])  
   
 # Generate predictions  
 output = model.forward(inputs, inputs, look\_ahead\_mask=look\_ahead\_mask)  
   
 # Get predicted tokens  
 predictions = np.argmax(output, axis=-1)  
   
 # Count correct predictions  
 for i in range(predictions.shape[^1]):  
 if predictions[0, i] == targets[0, i]:  
 correct\_predictions += 1  
 total\_predictions += 1  
   
 accuracy = correct\_predictions / total\_predictions  
 print(f"Test Accuracy: {accuracy:.4f}")  
   
 # Generate some text for qualitative evaluation  
 generate\_text(model, "Once upon a time", 100, char\_to\_idx, idx\_to\_char)  
   
 return accuracy  
  
def generate\_text(model, seed\_text, length, char\_to\_idx, idx\_to\_char):  
 """  
 Generate text using the trained model.  
   
 Parameters:  
 - model: Trained transformer model  
 - seed\_text: Initial text to start generation  
 - length: Length of generated sequence  
 - char\_to\_idx: Character to index mapping  
 - idx\_to\_char: Index to character mapping  
   
 Returns:  
 - Generated text  
 """  
 # Tokenize seed text  
 input\_tokens = [char\_to\_idx[ch] for ch in seed\_text]  
   
 # Generate tokens  
 generated\_text = seed\_text  
   
 for \_ in range(length):  
 # Convert input to array  
 inputs = np.array([input\_tokens])  
   
 # Create mask  
 look\_ahead\_mask = create\_look\_ahead\_mask(inputs.shape[^1])  
   
 # Generate prediction  
 output = model.forward(inputs, inputs, look\_ahead\_mask=look\_ahead\_mask)  
   
 # Get next token  
 next\_token = np.argmax(output[0, -1])  
   
 # Add token to sequence  
 input\_tokens.append(next\_token)  
 generated\_text += idx\_to\_char[next\_token]  
   
 print("Generated Text:")  
 print(generated\_text)  
   
 return generated\_text

**Complete Pre-training Pipeline**

Here's how to put it all together[[10]](#fn10)[[11]](#fn11):

def train\_and\_test\_transformer():  
 """  
 Complete pipeline for training and testing a transformer model.  
 """  
 # Load your text corpus  
 with open("corpus.txt", "r") as f:  
 text\_corpus = f.read()  
   
 # Parameters  
 vocab\_size = 5000  
 d\_model = 128  
 num\_heads = 8  
 d\_ff = 512  
 num\_layers = 4  
 max\_seq\_length = 100  
   
 # Prepare data  
 sequences, char\_to\_idx, idx\_to\_char = prepare\_data(text\_corpus, vocab\_size, max\_seq\_length)  
   
 # Split into train and test sets  
 split = int(0.9 \* len(sequences))  
 train\_sequences = sequences[:split]  
 test\_sequences = sequences[split:]  
   
 # Initialize model  
 model = Transformer(  
 vocab\_size=len(char\_to\_idx),  
 d\_model=d\_model,  
 num\_heads=num\_heads,  
 d\_ff=d\_ff,  
 num\_layers=num\_layers,  
 max\_seq\_length=max\_seq\_length  
 )  
   
 # Pre-train model  
 model, losses = pre\_train(  
 model=model,  
 sequences=train\_sequences,  
 epochs=10,  
 batch\_size=32,  
 learning\_rate=0.001  
 )  
   
 # Test model  
 test\_accuracy = test\_model(model, test\_sequences, char\_to\_idx, idx\_to\_char)  
   
 return model, losses, test\_accuracy

**Conclusion**

Building neural networks and transformers from scratch provides a deep understanding of their internal mechanisms. While frameworks like TensorFlow and PyTorch offer optimized implementations, implementing these models from scratch is an invaluable learning experience.

This guide has covered:

1. **Neural Network Implementation**: We built a complete neural network with forward and backward propagation, enabling efficient learning from data.
2. **Transformer Architecture**: We implemented the core components of transformers, including self-attention, multi-head attention, and positional encoding.
3. **Pre-training and Testing**: We developed methods for training and evaluating our models on language modeling tasks.

For practical applications, you would typically use established libraries that provide optimized implementations. However, this from-scratch implementation offers insights into the fundamental principles underlying these powerful models.

⁂

1. <https://eisenjulian.github.io/deep-learning-in-100-lines/index.html>

1. <https://machinelearningmastery.com/implement-backpropagation-algorithm-scratch-python/>

1. <https://www.reddit.com/r/learnmachinelearning/comments/1dv22tw/how_to_build_a_simple_neural_network_from_scratch/>

1. <https://www.digitalocean.com/community/tutorials/constructing-neural-networks-from-scratch>

1. <https://github.com/Veeransh14/Transformer-From-Scratch>

1. <https://github.com/retrogtx/attention-is-all-you-need>

1. <https://github.com/jsbaan/transformer-from-scratch>

1. <https://mayankblogs.hashnode.dev/build-your-own-transformer-model-from-scratch-using-pytorch>

1. <https://www.machinelearningmastery.com/a-gentle-introduction-to-positional-encoding-in-transformer-models-part-1/>

1. <https://www.kaggle.com/code/arnabs007/pretrain-a-bert-language-model-from-scratch>

1. <https://huggingface.co/learn/nlp-course/en/chapter7/6>