Tokens-to-Thought: A Contextual Transformer

A Four-Week Journey from NumPy to Transformers

Amogh Agrawal
Roll No. 24b1092

February 2026

Abstract

This report documents my progression through a four-week deep learning course, starting from fundamental Python scientific computing and culminating in the implementation of a character-level Transformer model for Shakespearean text generation. Each week built upon the previous, creating a coherent path from array manipulations to attention mechanisms. The hands-on approach of implementing concepts from scratch before using frameworks proved invaluable for developing intuition about how neural networks learn.

Contents

1 Introduction

The goal of this course was ambitious: understand deep learning deeply enough to implement a Transformer from scratch. Rather than jumping straight to PyTorch tutorials, we started with the building blocks—NumPy arrays, gradient descent, and single neurons—before gradually adding complexity.

This bottom-up approach meant that by the time I reached the Transformer implementation in Week 4, concepts like backpropagation and vectorized operations felt natural rather than magical. The journey can be summarized as:

Week	Focus
1	Scientific Python: NumPy, Matplotlib, Gradient Descent
2	Neural Networks from Scratch (NumPy only)
3	TensorFlow: Built-in and Custom Layers
4	Transformer Architecture for Text Generation

2 Week 1: Python Scientific Computing

2.1 NumPy Fundamentals

The first week established fluency with NumPy, Python's cornerstone library for numerical computation. Rather than simply calling functions, I focused on understanding *why* certain patterns work.

2.1.1 Array Initialization and Reshaping

NumPy's power lies in its ability to treat arrays as mathematical objects. I practiced multiple initialization patterns:

```
# Creating a 2x3 matrix via reshape
arr = np.array([1, 2, 4, 7, 13, 21]).reshape(2, 3)

# Using modern RNG for reproducibility
rng = np.random.default_rng(seed=42)
x = rng.random((n_rows, n_columns))

# Broadcasting to create constant arrays
zeros = np.broadcast_to(np.array(0), (4, 5, 2)).copy()
ones = np.full((4, 5, 2), 1)
```

The key insight was that `reshape` doesn't copy data—it creates a new view of the same memory, which is both efficient and sometimes surprising.

2.1.2 Vectorization

The most important lesson from Week 1 was the dramatic performance difference between loops and vectorized operations:

Approach	Time (1000×1000 array)
Nested Python loops	∼95 ms
NumPy vectorized	∼0.8 ms

This 100× speedup becomes critical when training neural networks on millions of samples.

2.2 Data Visualization

Using company sales data, I completed three visualization exercises that emphasized Matplotlib's object-oriented API over the stateful pyplot interface:

1. **Line plot**: Monthly profit trends with styled markers and legends
2. **Multi-line plot**: Product comparison using colormaps
3. **Pie chart**: Annual sales distribution with percentage labels

2.3 Multivariate Gradient Descent

The week concluded with implementing gradient descent to minimize:

$$f(x, y) = x^4 + x^2y^2 - y^2 + y^4 + 6$$

The analytical gradient is:

$$\nabla f = \begin{pmatrix} 4x^3 + 2xy^2 \\ 4y^3 - 2y + 2x^2y \end{pmatrix}$$

My implementation included backtracking line search—if a step increased the objective, the learning rate was halved until improvement occurred. This prevented divergence from aggressive step sizes.

3 Week 2: Neural Networks from Scratch

Week 2 was the conceptual heart of the course: implementing feedforward networks using only NumPy. No TensorFlow, no sklearn—just arrays and calculus.

3.1 The Perceptron

A perceptron computes:

$$\hat{y} = \sigma\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

where σ is a step function for classification. I implemented this with the perceptron learning rule: update weights only on misclassification.

3.1.1 AND Gate: Success

The AND gate is linearly separable, so a single perceptron learns it perfectly:

x_1	x_2	AND
0	0	0
0	1	0
1	0	0
1	1	1

3.1.2 XOR Gate: Failure

XOR cannot be represented by a single linear boundary:

x_1	x_2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

No matter how long training runs, the perceptron cannot solve XOR. This motivated the need for hidden layers.

3.2 Two-Layer Network

To overcome the linear separability limitation, I implemented a network with one hidden layer:

$$\begin{aligned}\mathbf{z}^{[1]} &= \mathbf{W}^{[1]}\mathbf{x} + \mathbf{b}^{[1]} \\ \mathbf{a}^{[1]} &= \tanh(\mathbf{z}^{[1]}) \\ \mathbf{z}^{[2]} &= \mathbf{W}^{[2]}\mathbf{a}^{[1]} + \mathbf{b}^{[2]} \\ \hat{y} &= \sigma(\mathbf{z}^{[2]})\end{aligned}$$

3.2.1 Backpropagation

The gradients flow backward through the network via the chain rule:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{[2]}} &= \frac{1}{m}(\mathbf{a}^{[1]})^T \cdot (\hat{y} - y) \\ \frac{\partial \mathcal{L}}{\partial \mathbf{W}^{[1]}} &= \frac{1}{m}\mathbf{x}^T \cdot \left[(\hat{y} - y) \cdot (\mathbf{W}^{[2]})^T \odot (1 - (\mathbf{a}^{[1]})^2)\right]\end{aligned}$$

The term $(1 - \mathbf{a}^2)$ is the derivative of tanh, and $\odot$ denotes element-wise multiplication.

3.3 Logic Circuits

With the two-layer network, I successfully implemented:

1. **XOR gate**: 4 hidden units, perfect accuracy after 5000 epochs
2. **Full adder**: 6 hidden units, learns sum and carry outputs
3. **Ripple-carry adder**: Composes full adders for multi-bit addition

The ripple-carry adder was particularly satisfying—watching a neural network correctly compute $13 + 11 = 24$ by propagating carries through learned full adders.

4 Week 3: TensorFlow Fundamentals

Having built networks from scratch, Week 3 introduced TensorFlow as a practical framework. The goal was to understand what happens inside `Dense` and `Flatten` layers.

4.1 MNIST Classification

The MNIST dataset consists of 28×28 grayscale images of handwritten digits. A simple feed-forward network achieves impressive accuracy:

```
model = Sequential([
    Flatten(input_shape=(28, 28)),
    Dense(128, activation='relu'),
    Dense(10, activation='softmax')
])
```

Result: 97.2% test accuracy after 5 epochs.

4.2 Custom Layer Implementation

To demystify TensorFlow's layers, I implemented custom versions:

```
class CustomDenseReluLayer(tf.keras.layers.Layer):
    def __init__(self, units):
        super().__init__()
        self.units = units

    def build(self, input_shape):
        self.w = self.add_weight(
            shape=(input_shape[-1], self.units),
            initializer='glorot_uniform',
            trainable=True
        )
        self.b = self.add_weight(
            shape=(self.units,),
            initializer='zeros',
            trainable=True
        )

    def call(self, inputs):
        return tf.nn.relu(tf.matmul(inputs, self.w) + self.b)
```

The custom model achieved identical accuracy to the built-in version, confirming my implementation was correct.

4.3 Housing Price Regression

For the regression assignment, I compared linear and non-linear models on the California Housing dataset:

Model	Test MSE	Test MAE
Linear Regression (1 neuron)	0.52	0.53
Feedforward NN (64→32→16→1)	0.26	0.35

The neural network's ability to learn non-linear relationships cut the MSE roughly in half.

5 Week 4: Transformer Architecture

The final week was the culmination of everything: implementing a Transformer model to generate Shakespearean text. This required understanding attention mechanisms, positional encodings, and autoregressive generation.

5.1 Problem Setup

Given a corpus of Shakespeare's works (∼40,000 lines), train a character-level language model that predicts the next character given a context window. The model learns to generate text that mimics Shakespeare's style.

5.2 Architecture Overview

The model consists of:

1. Token + Position Embeddings
2. 4 Transformer Blocks
3. Linear projection to vocabulary logits

5.2.1 Positional Encoding

Unlike RNNs, Transformers have no inherent notion of sequence order. Positional embeddings inject this information:

$$\mathbf{e}_{\text{input}} = \mathbf{E}_{\text{token}}[x] + \mathbf{E}_{\text{pos}}[\text{position}]$$

Both embeddings are learned during training.

5.2.2 Causal Self-Attention

The key innovation of Transformers is self-attention, which allows each position to attend to all others. For autoregressive generation, we apply a causal mask ensuring position i only attends to positions $\leq i$:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} + M\right)V$$

where M is a mask with $-\infty$ for future positions.

```python
def causal_attention_mask(batch_size, n_dest, n_src):
    mask = tf.linalg.band_part(tf.ones((n_dest, n_src)), -1, 0)
    return tf.cast(mask, dtype=tf.bool)
```

5.2.3 Transformer Block

Each block contains:

1. Multi-head self-attention (8 heads)
2. Residual connection + Layer normalization
3. Feed-forward network (GELU activation)
4. Residual connection + Layer normalization

5.3 Training

Hyperparameter	**Value**
Context length (block size)	128
Embedding dimension	256
Attention heads	8
Transformer blocks	4
Dropout	0.1
Batch size	32
Epochs	15

5.4 Text Generation

Generation proceeds autoregressively:

1. Start with a seed sequence of length `block_size`
2. Run forward pass to get next-token probabilities
3. Sample from the distribution (temperature-controlled)
4. Append sampled token, shift window, repeat

Temperature (τ) controls randomness:

$$p_i = \frac{\exp(z_i/\tau)}{\sum_j \exp(z_j/\tau)}$$

- $\tau = 0.5$: Conservative, more repetitive
- $\tau = 0.8$: Balanced creativity
- $\tau = 1.0$: More varied output

5.5 Sample Output

Prompt: "To be or not to be"
Generated ($\tau = 0.7$):

> *To be or not to be the cause of all my heart, And therefore I will not be so much as I am, For I have seen the time that I have been a man of such a nature that I have no more to say...*

While not perfect, the model captures Shakespearean cadence and vocabulary.

6 Reflections and Key Takeaways

6.1 What Worked Well

1. **Bottom-up learning**: Implementing backpropagation from scratch made TensorFlow's abstractions meaningful rather than magical.

2. **Incremental complexity**: Each week built directly on the previous, creating a coherent narrative.

3. **Concrete exercises**: Logic gates and adders provided immediate feedback on whether implementations were correct.

6.2 Challenges Faced

1. **Numerical instability**: Early gradient descent implementations diverged until I added gradient clipping and careful initialization.

2. **Debugging attention**: The causal mask was tricky to get right—off-by-one errors led to information leakage.

3. **Training time**: The Transformer required GPU acceleration; CPU training was prohibitively slow.

6.3 Future Directions

This course provided a foundation for several next steps:

- Implementing word-level rather than character-level models
- Exploring different positional encoding schemes (sinusoidal, RoPE)
- Training larger models with more data
- Fine-tuning pre-trained models for specific tasks

7 Conclusion

Over four weeks, I progressed from basic array operations to implementing a working Transformer. The journey reinforced that deep learning, despite its complexity, rests on surprisingly simple foundations: matrix multiplication, differentiation, and iterative optimization.

The most valuable insight was understanding *why* things work, not just *how* to call APIs. When the Transformer finally generated coherent Shakespearean prose, it felt less like magic and more like the natural consequence of the principles learned in Weeks 1 through 3.

Repository: `https://github.com/amoghagrawal/Tokens-to-Thought-A-Contextual-Transformer`