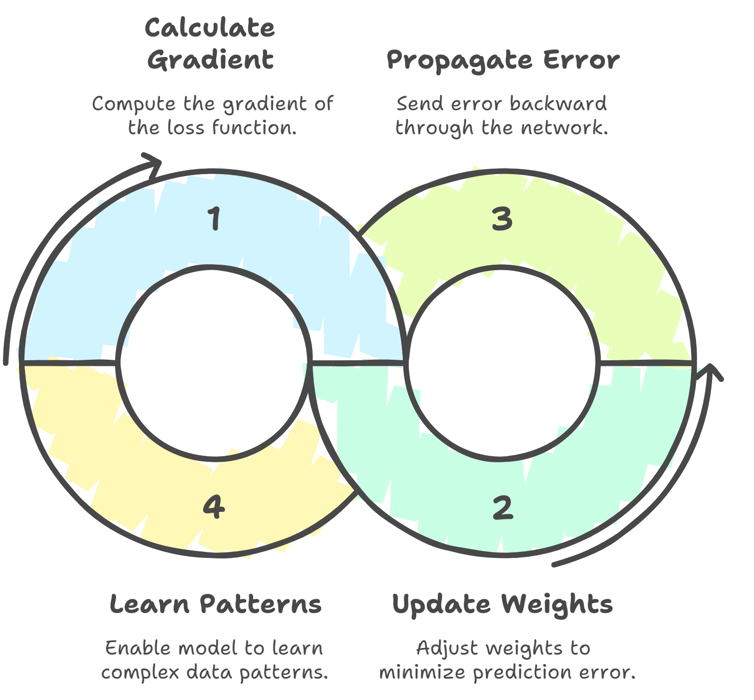
Backpropagation Cycle in Transformers



Transformer Decoder Process Flow

1. ✅ The decoder processes one token at a time. Let's use the case for the token "Le", and its decoder output is:

```
Z = [1.07006, 0.95765]
```

2. ✅ The decoder includes:

Self-attention (masked)
Add & Norm
Cross-attention
Add & Norm
Feedforward network (FFN)
Add & Norm
Linear projection to vocabulary
Softmax → Loss

We'll now walk through backpropagation for each.

3. ✅ Final decoder output: z = [1.07006, 0.95765]

Vocabulary projection matrix: **W_vocab =**

```
[0.8   0.2
 0.1   0.5
-0.2   0.4]
```

logits = z × W_vocab^T = [1.0476, 0.5858, 0.1690]

Softmax : ŷ = [1.0476, 0.5858, 0.1690]

Target: "chat" (index = 1), so:

```
y = [0, 1, 0]
Loss = -log(0.308) ≈ 1.176
```

4. ✅ Linear Projection → Softmax

Step 1: Gradient of loss w.r.t. logits: ∂logits/∂L = y_hat − y = [0.489, −0.692, 0.203]

Step 2: Gradient of loss w.r.t. W_vocab: ∂L/∂W_vocab = z^T ⊗ ∂logits/∂L

```
= [1.07006*0.489, 1.07006*(−0.692), 1.07006.0.203]  =  [0.523, −0.741, 0.217
   0.95765*0.489, 0.95765*(−0.692), 0.95765.0.203]      0.468, −0.662, 0.194]
```

Step 3: Gradient w.r.t. decoder output z:

```
                                                       [0.8   0.2
∂L/∂z = ∂logits/∂L * W_vocab =[1.0476, 0.5858, 0.1690] * 0.1   0.5
                                                       -0.2   0.4]

      = [0.281, −0.167]
```

5. ✅ Decoder FFN

FFN uses two layers:

W1 ∈ R^ (2× 4)
W2 ∈ R^ (4×2)

Given: Decoder input to FFN: x= [0.5, 0.5]

From the forward pass:

```
z1 = x @ W1 = [0.4, 0.1, 0.35, -0.05]
a1 = ReLU(z1) = [0.4, 0.1, 0.35, 0]
z2 = a1 @ W2 = z (→ passed to projection)
```

Step 1: Backprop through Linear 2: ∂L/∂W_2 = a1^T ⊗ ∂L/∂z

```
  [0.40*0.281, 0.40*(−0.167)       [0.1124, −0.0668]
=  0.10*0.281, 0.10*(−0.167)    =   0.0281, −0.0167
   0.35*0.281, 0.35*(−0.167)        0.0984, −0.0585
   0.00*0.281, 0.00*(−0.167)]       0.0000,  0.0000]
```

Step 2: Backprop to ReLU: ∂L/∂a1 = ∂L/∂z . W_2^T

```
= [0.281, −0.167] . W_2^T
= [0.146, 0.001, 0.207, −0.162]
```

```
Apply ReLU derivative (0 if z1 < 0): ∂L/∂z1=[0.146,0.001,0.207,0]
```

Step 3: Gradient of : `W1 = ∂L/∂W1 = x^T ⊗ ∂L/∂z1`

```
[0.5*0.146, 0.5*0.001, 0.5*0.207, 0] = [0.073, 0.0005, 0.1035, 0
[0.5*0.146, 0.5*0.001, 0.5*0.207, 0]    0.073, 0.0005, 0.1035, 0]
```

6. ✅ Cross-Attention (Decoder → Encoder)

Decoder query: Qd=[1.204,0.895]

Encoder keys: K1=[4.689,1.905]
K2=[4.818,2.409]

Softmax attention weights: α=[0.507,0.493]

Output of cross-attention: o = α1.V1 + α2.V2 = [5.0918,2.3458]

Let gradient from FFN be: ∂L/∂o = [0.281, -0.167]

Step 1: Gradient w.r.t. values (V):

```
∂L/∂V1 = α1 ⊗ ∂L/∂o
= [0.507] . [0.281, -0.167]
= [0.1425, −0.0847]
```

```
∂L/∂V2 = α2 ⊗ ∂L/∂o
= [0.493] . [0.281, -0.167]
= [0.1385, −0.0823]
```

Step 2: Gradient w.r.t. attention weights:

```
∂L/∂α1 = ∂L/∂o.V1
= [0.281*4.689 − 0.167*1.905]
≈ 0.9994
```

```
∂L/∂α2 = ∂L/∂o.V2
= [0.281*4.818 − 0.167*2.409]
≈ 0.9515
```

Use Softmax Jacobian for:

```
∂score/∂Q
∂score/∂K

Then:        ∂L/∂Q = ∑∂L/∂αi * Ki

             ∂L/∂Ki = ∑∂L/∂αi * Qd
```

7. ✅ Decoder Self-Attention

From PDF:

```
Single token: x = [0.5,0.5]
x = [0.5,0.5]

Wq = I, Wk = [[1, 0], [1, 1]],
Wv = [[1.1, 0], [0, 1.1]]
Output = V (since attention score is 1.0)
```

Let d_out = [dz1 from FFN]

```
= [0.281, -0.167]
```

```
Then:

∂L/∂V = dout
∂L/∂Wv = Xt . dout = [0.5, 0.5]^T ⊗ [0.281, -0.167]
```

8. ✅ **FINAL GRADIENT UPDATES**

```
Assume learning rate η = 0.1:

Projection:        W_vocab ← W_vocab −0.1*∂W_vocab

                   W_1 ← W_1 − 0.1*∂W_1
Decoder FFN:       W_2 ← W_2 − 0.1*∂W_2

Cross-Attention:   W_1 ← W_1 − 0.1*∂W_1
                   W_2 ← W_2 − 0.1*∂W_2
```

Transformer Encoder Process Flow

1. ✅ **Encoder Backpropagation**

From decoder's cross-attention, we had:

Query from decoder: Q_d =[1.204, 0.895]

Encoder key-value pairs: K1 = [4.689, 1.905]
K2 = [4.818, 2.409]

Attention weights: α1 = [0.507,0.493

Output : o = α1 * V1 + α2 * V2 =[5.0918,2.3458]

Gradient flowing into encoder values (from decoder) :

```
∂L/∂o = [0.281, −0.167]
```

Step 1: Gradient w.r.t. Encoder Values (V):

We already computed:

```
dL/dV1 = α1 * ∂L/∂o = 0.507*[0.281,−0.167] = [0.1425, −0.0847]
dL/dV2 = α2 * ∂L/∂o = 0.493*[0.281,−0.167] = [0.1385, −0.0823]
```

Step 2: Gradient w.r.t. Encoder Keys (K):

We decoder:

```
dL/dα1 = ∂L/∂o * V1 = 0.281*4.689 − 0.167*1.905 = 1.318
dL/dα2 = ∂L/∂o * V2 = 0.281*4.818 − 0.167*2.409 = 1.26
```

Assuming softmax with two scores s1 , s2, we apply the Jacobian:

```
Let α1 =0.507, α2 =0.493
```

Then softmax gradients w.r.t. scores are:

```
dL/ds1 = dα1.α1(1-α1) - dα2.α1α12
```

Now backprop through:

```
si = Q_d.K_i^T/ square_root(d_k)

=> dL/dK_i = (Q_d . dL/ds_i)/square_root(d_k)
```

Step 3: Gradients w.r.t. Encoder Self-Attention:

Each encoder token attends to others: `Q = X.Wq, K=X.Wk, V=X.Wv`

Attention output for token_i : `∑jSoftmax(Q_iK_j^T)V_j`

We apply:

```
dL/dV = αt * d_out
dL/dV = Qt * d_score
dL/dQ = d_score * K
```

Then Compute:

```
dWq = Xt * dQ
dWk = Xt * dK
dWv = Xt * dV
```

Step 4: Encoder Feedforward Network:

Assume input to encoder FFN is x = [1.1, 1.1] for token 3:

Forward:

```
z1 = x*W1 = [0.88,0.22,0.77,−0.11]
a1 = ReLU(z1) = [0.88,0.22,0.77,0]
z2 = a1.W2 = passed to next layer
```

Backward:

```
dL/dz2 = [0.281,−0.167] ⇒ dW2 = a1t*dz2
da1 = dz2.W2t, dz1 = da1.ReLU'(z1)
dW1 = xt.dz1
```

Step 5: Encoder LayerNorm + Residuals:

Let input to LayerNorm = x

Mean: μ `∑(xi^2)/d`

Variance: σ2`∑((xi−μ)^2)/d`

Backprop:

```
dx_norm = γ * δ
```

Use: `dx = [δ−mean(δ)−x_norm*mean(δ*x_norm)]/sqrt(δ + E)`

Update LayerNorm weights:

```
dγ = ∑δ*x_norm
dβ = ∑δ
```

Step 6: Final Encoder Weight Updates:

With learning rate `η=0.1: W_new = W_old −0.1(dL/dW)`