Fusion Variations

🔹 **1. Concatenation (Baseline)**
What it is:
Combines the news and metric embeddings by placing them side by side into a single vector. No interaction is learned — it's a raw merge of features.

Difference from baseline:
This is the baseline — simple, static, and parameter-free. It does not model the relationship between news and metrics, nor does it adapt across samples. The model must learn all interactions in downstream layers.

How/Where to implement:
After extracting a `[batch, 9, 128]` news embedding and a `[batch, 9, 384]` metric embedding, concatenate along the last dimension → `[batch, 9, 512]`. This tensor goes directly into the transformer encoder or MLP head.

Why it's useful for this project:
Concatenation serves as a strong, interpretable baseline. It ensures that both modalities are fully preserved before the model processes them. This is useful when validating whether more complex fusion methods yield meaningful improvements. For instance, if Apple's price drops 2% on Day 5 and the news reads "Apple faces antitrust lawsuit," this fusion simply forwards both inputs to the model. While it doesn't relate them, it gives the model full access to the raw features from both sides — news and metrics — allowing learning in later stages.

🔹 **2. Gated MLP (Learned Fusion)**
What it is:
Learns a dynamic weighting between news and metric embeddings. A small neural network decides how much of each modality should contribute to the fused representation.

Difference from baseline:
Unlike static concatenation, which gives equal weight to both modalities, Gated MLP allows the model to adaptively emphasize news or metrics depending on the input. This is especially helpful in scenarios where one modality is more informative than the other — something the baseline cannot model.

How/Where to implement:
Add a 1–2 layer MLP after both embeddings are extracted. Use it to compute a gate vector α. The fusion is:
fused = α * news + (1 - α) * metrics.

Why it's useful for this project:
This approach mimics how real-world analysts prioritize different signals depending on context. For example, during Amazon's earnings on Day 3, strong earnings reports in the news may lead the model to increase α, relying more on the news embedding. On Day 4, when there's little headline activity, the model can shift α to favor the historical metrics. Gated MLP provides this adaptability, improving decision quality on volatile or information-heavy days — something concatenation cannot do.

Metric Encoder Variations

🔹 **1. Group-wise 1D Convolution (Baseline)**
What it is:
Applies 1D convolution across 9 days of input for each of the 6 financial metrics separately, with 5 stocks treated as channels per metric. The convolution is grouped, meaning each metric is processed independently.

Difference from baseline:
This is the baseline — it treats each metric separately and captures local trends across short windows (e.g., 3-day fluctuations). However, it cannot model long-term dependencies or interactions between days that are far apart.

How/Where to implement:
Use PyTorch nn.Conv1d with groups=6 and input shape [batch, 30, 9] (6 metrics × 5 stocks). Output shape becomes [batch, 384] per day (with 64 filters per metric). Pass through a nonlinearity like ReLU before fusion.

Why it's useful for this project:
Grouped convolutions are effective at capturing short-term movements, such as sharp rebounds or sudden dips. For example, if a "V"-shaped recovery is observed over 3 days — [180, 178, 181] — the kernel can detect this as a reversal pattern. This aligns with common technical indicators like RSI and moving averages that traders use to detect momentum or overbought/oversold conditions.

🔹 **2. LSTM over Metrics**
What it is:
Applies a recurrent LSTM over the 9-day sequence of each financial metric (or over the flattened metrics per day), allowing the model to learn temporal dependencies.

Difference from baseline:
Unlike convolution, which captures local patterns, LSTM tracks longer-term trends and sequence dependencies. This enables it to model delayed effects or gradual build-up that 1D CNNs can miss.

How/Where to implement:
Reshape the metric tensor into [batch, 9, 30] and feed it into nn.LSTM. Use either the final hidden state or the full output sequence as the metric representation.

Why it's useful for this project:
In financial markets, responses to signals are often delayed. For instance, rising volume over Days 1–6 may precede a price breakout on Day 7. An LSTM can recognize this trend, where a convolution may miss the broader buildup. This enables more effective modeling of momentum shifts, accumulation phases, and mean-reversion signals.

🔹 **3. TCN (Temporal Convolutional Network)**
What it is:
A 1D CNN architecture that uses dilated convolutions to model long-range dependencies while preserving sequence length.

Difference from baseline:
TCN expands the model's temporal receptive field without using recurrence. Unlike Conv1D, which only sees short windows (e.g., 3 days), TCN can incorporate all 9 days with stacked dilations. It also allows for parallelization, unlike RNNs.

How/Where to implement:
Replace Conv1D with dilated convolutions using increasing dilation rates (1 → 2 → 4). Use causal padding to maintain [batch, 9, d] shape.

Why it's useful for this project:
TCNs are well-suited for detecting gradual momentum or trend reversals. For example, if prices slowly build over 9 days, TCN can detect that long-range build-up — something

a small CNN might miss due to its limited window. It balances efficiency and memory, ideal for low-resource training environments like Colab.

Prediction Head Variations

🔹 **1. MLP with ReLU (Baseline)**
What it is:
A simple multi-layer perceptron (MLP) that takes the fused representation and outputs a scalar prediction for Day 10 — either the closing price or direction (up/down). The model uses non-linear activations such as ReLU.

Difference from baseline:
This is the baseline. It assumes that all useful information — including temporal dynamics — has already been encoded in the input. It treats the sequence as a static set of features after pooling (e.g., mean or max over the 9-day window), thus ignoring temporal order. It cannot model evolving trends or delayed reactions.

How/Where to implement:
After obtaining the fused `[batch, 9, 312]` tensor, apply mean or max pooling along the time axis to get `[batch, 312]`. Then pass it through a 2–3 layer MLP, such as `312 → 128 → 1`, with optional dropout or normalization layers.

Why it's useful for this project:
This is ideal for fast prototyping or low-resource training. It can perform well when market signals are consistent across time — for example, if sentiment is steadily positive for all 9 days and price steadily rises, pooling followed by MLP is enough. However, it cannot detect when things changed — e.g., if prices reversed on Day 7, the MLP won't capture this turning point.

🔹 **2. GRU/LSTM on Fused Sequence**
What it is:
A recurrent layer (GRU or LSTM) applied over the fused 9-day sequence. It learns how daily news and metric interactions evolve across time and captures sequential dependencies.

Difference from baseline:
Unlike the static MLP, this approach maintains temporal order and memory, tracking how the fused features change from day to day. It can model delayed effects, compounding sentiment, and sudden shifts — capabilities the MLP baseline lacks.

How/Where to implement:
Feed the fused tensor [batch, 9, 312] into nn.GRU or nn.LSTM with hidden size 128–256. Use the final hidden state or attention pooling over the sequence to produce a [batch, hidden_dim] vector. Feed this into an output MLP for regression or classification.

Why it's useful for this project:
Market behavior often unfolds across several days. For example, if a policy rumor appears on Day 3 but price reacts on Day 6, an LSTM can connect the cause and effect through memory. Similarly, if news sentiment is building positively over time, the GRU can learn this rising trend and reflect it in its output. This is particularly helpful in volatile conditions or narrative-driven markets where timing matters.

Efficient Tuning Techniques

🔹 **1. Soft Prompting**
What it is:
Learns a set of continuous embeddings (soft prompts) that are prepended to the input sequence. These act like additional tokens guiding the model, while keeping the transformer weights completely frozen.

Difference from baseline (vs. LoRA or full fine-tuning):
Soft prompts do not modify the model's parameters or attention layers. They simply add a small number of trainable vectors. Compared to LoRA (which updates weight projections) or full fine-tuning (which updates everything), soft prompts offer lightweight, memory-efficient adaptation — ideal for small datasets and limited compute.

How/Where to implement:
Add a learnable tensor of shape [prompt_length, hidden_dim] (e.g., [5, 312]) to the beginning of the fused input sequence. This becomes [batch, 9 + prompt_length, 312], which is then passed into the frozen transformer.

Why it's useful for this project:
This approach is ideal when compute or memory is limited, like when training on Google Colab. It also allows fast experimentation without risk of overfitting. For example, prompts can encode task-specific cues like "focus on volatility," "treat news as primary," or "highlight trend continuity," giving the frozen model a way to adapt without touching the backbone. This helps preserve generalization while improving specialization.