Transforming Equations to Latex

Nitesh Choubey(2021ME21057) Jatin Hyanki(2021ME21117)

MCL 775 : Special Topics in Industrial Engineering

Presentation Overview

Objective

- **Task**: Convert images of mathematical expressions into LaTeX code.

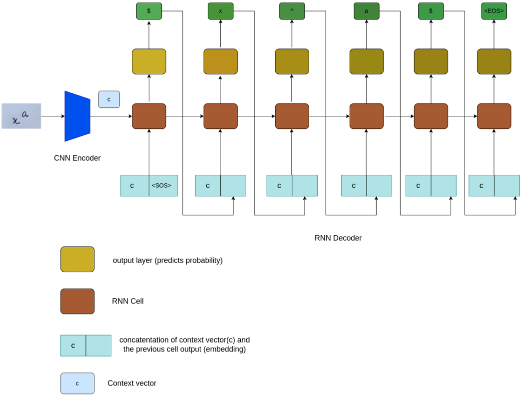


Figure: Model Architecture

Data transformation

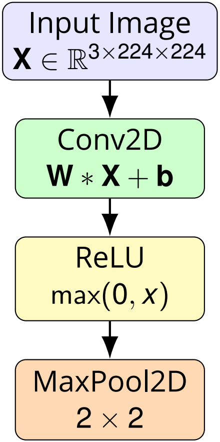
```python
# Dataset class
class CustomDataset(Dataset):
    def __init__(self, csv_file, image_dir, transform=None, maxSamples=None):
        self.data = pd.read_csv(csv_file)
        if maxSamples is not None:
            self.data = self.data.head(maxSamples)
        self.image_dir = image_dir
        self.transform = transform

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        img_name = self.data.iloc[idx, 0]
        img_path = os.path.join(self.image_dir, img_name)
        try:
            image = Image.open(img_path).convert('RGB')
        except Exception as e:
            print(f"Error loading image {img_path}: {e}")
            # Return a black image if loading fails
            image = Image.new('RGB', (224, 224), (0, 0, 0))

        formula = self.data.iloc[idx, 1]
        formula_split = formula.strip().split()

        if self.transform:
            image = self.transform(image)

        return image, formula_split

# Improved image transformations
train_transforms = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.RandomRotation(2),  # makes model more robust
    transforms.ColorJitter(brightness=0.1, contrast=0.1),  # increasing the robustness by changing brightness randomly
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])  # ImageNet stat's mean and std deviation
])

# Test transforms without augmentation
test_transforms = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])  # ImageNet stats
])
```

Figure: Transformation

```python
# Build vocabulary
vocab = set()
for _, formula in train_data:
    vocab.update(formula)

# Token mappings
word_to_index = {word: i+3 for i, word in enumerate(vocab)}
word_to_index["<pad>"] = 0
word_to_index["<sos>"] = 1
word_to_index["<unk>"] = 2
word_to_index["<eos>"] = len(word_to_index)

index_to_word = {idx: word for word, idx in word_to_index.items()}
vocab_size = len(word_to_index)
print(f"Vocabulary size: {vocab_size}")

# Improved collate function
def collate_fn(batch):
    images, formulas = zip(*batch)

    # Handle bad images - replace with black image if needed
    valid_samples = []
    for i, img in enumerate(images):
        if img is not None and not torch.isnan(img).any():
            valid_samples.append(i)

    if len(valid_samples) < len(images):
        print(f"Warning: {len(images) - len(valid_samples)} invalid images removed from batch")
        images = [images[i] for i in valid_samples]
        formulas = [formulas[i] for i in valid_samples]

    if not images:  # If no valid images in batch
        return None, None

    images = torch.stack(images)

    sequences = []
    for formula in formulas:
        seq = [word_to_index["<sos>"]]
        for word in formula:
            seq.append(word_to_index.get(word, word_to_index["<unk>"]))
        seq.append(word_to_index["<eos>"])
        sequences.append(torch.tensor(seq))

    padded_sequences = pad_sequence(sequences, batch_first=True, padding_value=word_to_index["<pad>"])

    return images, padded_sequences
```

Figure: Transformation

Convolutional Block: Conv $\rightarrow$ *ReLU* $\rightarrow$ *Pooling*



Mathematics:

- **Convolution:**

$$y_{i,j,k} = \sum_{c=1}^{C} \sum_{m,n} W_{k,c,m,n} \cdot X_{c,i+m,j+n} + b_k$$

X: input, W: filter, b: bias, i, j: spatial pos, k: output channel, c: input channel, m, n: kernel offset, C: input channels total

- **ReLU:**

$$f(x) = \max(0, x)$$

- **Max Pooling:** Downsamples by taking max in 2×2 regions

Encoder

```python
# Dataset class
class CustomDataset(Dataset):
    def __init__(self, csv_file, image_dir, transform=None, maxSamples=None):
        self.data = pd.read_csv(csv_file)
        if maxSamples is not None:
            self.data = self.data.head(maxSamples)
        self.image_dir = image_dir
        self.transform = transform

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        img_name = self.data.iloc[idx, 0]
        img_path = os.path.join(self.image_dir, img_name)
        try:
            image = Image.open(img_path).convert('RGB')
        except Exception as e:
            print(f"Error loading image {img_path}: {e}")
            # Return a black image if loading fails
            image = Image.new('RGB', (224, 224), (0, 0, 0))

        formula = self.data.iloc[idx, 1]
        formula_split = formula.strip().split()

        if self.transform:
            image = self.transform(image)

        return image, formula_split

# Improved image transformations
train_transforms = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.RandomRotation(2),  # makes model more robust
    transforms.ColorJitter(brightness=0.1, contrast=0.1),  # increasing the robustness by changing brightness randomly
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])  # ImageNet stat's mean and std deviation
])

# Test transforms without augmentation
test_transforms = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])  # ImageNet stats
])
```

Figure: ResNet 34 based encoder

- Pretrained ResNet34 used for feature extraction.
- ResNet introduces **Residual Connections**, allowing input to bypass intermediate layers.
- Prevents the vanishing gradient.
- 33 convolutions layers
- 2 pooling layers.
- ReLU activation is applied after every convolutional layer.
- The architecture uses 16 residual blocks, each containing 2 convolutions, totaling 32 convolutions.
- AdaptiveAvgPool resizes output to fixed `(7, 7)` spatial size.
- Linear layer projects 512-dim features to desired `encoded_dim`.
- Mean-pooled context vector `(batch_size, encoded_dim)` used to initialize decoder.

Attention Mechanism

```python
# Attention mechanism
class Attention(nn.Module):
    def __init__(self, encoder_dim, decoder_dim, attention_dim):
        super(Attention, self).__init__()
        # mapping encoder dim to attention dim
        self.encoder_att = nn.Linear(encoder_dim, attention_dim)
        # mapping decoder dim to attention dim
        self.decoder_att = nn.Linear(decoder_dim, attention_dim)
        self.full_att = nn.Linear(attention_dim, 1)

    def forward(self, encoder_out, decoder_hidden):
        # encoder_out: (batch_size, num_pixels, encoder_dim)
        # decoder_hidden: (batch_size, decoder_dim)
        # Projecting output and input in attention dim
        att1 = self.encoder_att(encoder_out)  # (batch_size, num_pixels, attention_dim)
        att2 = self.decoder_att(decoder_hidden).unsqueeze(1)  # (batch_size, 1, attention_dim)
        # Relevance of the code to the image patch
        att = F.relu(att1 + att2)  # (batch_size, num_pixels, attention_dim)
        # Reducing attention dim to 1 i.e each pixel has a single value now
        att = self.full_att(att).squeeze(2)  # (batch_size, num_pixels)
        (variable) alpha: Any
        alpha = F.softmax(att, dim=1)  # (batch_size, num_pixels)

        context = (encoder_out * alpha.unsqueeze(2)).sum(dim=1)  # (batch_size, encoder_dim)
        return context, alpha
```

Figure: Attention Mechanism

- Project encoder outputs and decoder hidden state into the attention space.
- Add the projected vectors to compute relevance scores for each image region.
- Apply ReLU to introduce non-linearity.
- Compress each region's score into a scalar using a linear layer.
- Normalize scores using softmax to get attention weights over image regions.
- Compute the context vector as a weighted sum of image features using these attention weights.

Embedding Layer

What is an Embedding?

- Maps token indices (e.g. `\frac`, `x`) to dense vectors.
- Learns meaningful representations during training.

Embedding Overview

Aspect	Details
Form	Matrix $E \in \mathbb{R}^{V \times D}$
Token Embedding	Row E_i = embedding for token i
Initial Values	Random at start
Training	backpropagation
Usage	`embedding(token_index)` returns E_i
Goal	Capture similarity between tokens

Example: `\frac` $\rightarrow$ [0.12, -0.88, ..., 0.03]

LSTM: Long Short-Term Memory

What is an LSTM?

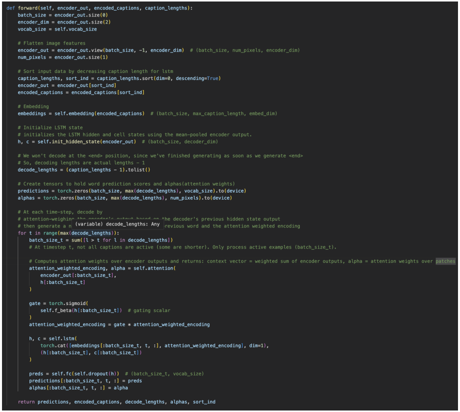
- A type of Recurrent Neural Network (RNN) designed to handle long-term dependencies.
- Maintains two states: hidden state (h_t) and cell state (C_t).
- Uses gates to control the flow of information.

LSTM Cell Components

Component	Role
h_t	Hidden state (short-term memory)
C_t	Cell state (long-term memory)
f_t	Forget gate: discard irrelevant info
i_t	Input gate: select what to add
$\tilde{C}_t$	Candidate memory: new proposed info
o_t	Output gate: decide final output

LSTM can learn what to remember, forget, and output.

Attention Decoder



```python
def forward(self, encoder_out, encoded_captions, caption_lengths):
    batch_size = encoder_out.size(0)
    encoder_dim = encoder_out.size(2)
    vocab_size = self.vocab_size

    # Flatten image features
    encoder_out = encoder_out.view(batch_size, -1, encoder_dim)  # (batch_size, num_pixels, encoder_dim)
    num_pixels = encoder_out.size(1)

    # Sort input data by decreasing caption length for lstm
    caption_lengths, sort_ind = caption_lengths.sort(dim=0, descending=True)
    encoder_out = encoder_out[sort_ind]
    encoded_captions = encoded_captions[sort_ind]

    # Embedding
    embeddings = self.embedding(encoded_captions)  # (batch_size, max_caption_length, embed_dim)

    # Initialize LSTM state
    # initializes the LSTM hidden and cell states using the mean-pooled encoder output.
    h, c = self.init_hidden_state(encoder_out)  # (batch_size, decoder_dim)

    # We won't decode at the <end> position, since we've finished generating as soon as we generate <end>
    # So, decoding lengths are actual lengths - 1
    decode_lengths = (caption_lengths - 1).tolist()

    # Create tensors to hold word prediction scores and alphas(attention weights)
    predictions = torch.zeros(batch_size, max(decode_lengths), vocab_size).to(device)
    alphas = torch.zeros(batch_size, max(decode_lengths), num_pixels).to(device)

    # At each time-step, decode by
    # attention-weighin                         the decoder's previous hidden state output
    # then generate a n (variable) decode_lengths: Any revious word and the attention weighted encoding
    for t in range(max(decode_lengths)):
        batch_size_t = sum([l > t for l in decode_lengths])
        # At timestep t, not all captions are active (some are shorter). Only process active examples (batch_size_t).

        # Computes attention weights over encoder outputs and returns: context vector = weighted sum of encoder outputs, alpha = attention weights over patches
        attention_weighted_encoding, alpha = self.attention(
            encoder_out[:batch_size_t],
            h[:batch_size_t]
        )

        gate = torch.sigmoid(
            self.f_beta(h[:batch_size_t])  # gating scalar
        )
        attention_weighted_encoding = gate * attention_weighted_encoding

        h, c = self.lstm(
            torch.cat([embeddings[:batch_size_t, t, :], attention_weighted_encoding], dim=1),
            (h[:batch_size_t], c[:batch_size_t])
        )

        preds = self.fc(self.dropout(h))  # (batch_size_t, vocab_size)
        predictions[:batch_size_t, t, :] = preds
        alphas[:batch_size_t, t, :] = alpha

    return predictions, encoded_captions, decode_lengths, alphas, sort_ind
```

Figure: foward function

- **Sort captions** by length.
- Convert LaTeX tokens to dense vectors using embedding layer.
- **Initialize hidden states** with average encoder output.
- **For each time-step** ***t*****:**
  - Compute attention weights and context vector.
  - Apply gating mechanism.
  - Concatenate embedding and attention-weighted encoding.
  - Pass through LSTM.
  - Output vocabulary scores.
- **Return:** Predictions, attention weights, sorted captions, and decode lengths.

Training Loop

Pseudocode (Training per Batch) [H]

(images, formulas) in loader Move data to device optimizer.zero_grad()
$(enc\_out, \mu) \leftarrow encoder(images)$
$(h, c) \leftarrow$ zeros(batch_size, hidden_size)
input $\leftarrow$ `<sos>` token for all samples loss $\leftarrow 0$
$t = 1$ sequence_length embed $\leftarrow embedding(input)$
$(context, \alpha) \leftarrow attention(enc\_out, h)$
$lstm\_in \leftarrow [embed \| context]$
$(h, c) \leftarrow LSTMCell(lstm\_in, (h, c))$ logits $\leftarrow out(h)$ output $\leftarrow$ log_softmax(logits)
loss += NLLLoss(output, target[:, t]) input $\leftarrow$ `teacher forcing or predicted token`

loss.backward() clip gradients of encoder and decoder optimizer.step()

Explanation

- **encoder:** Extracts spatial image features.
- **embedding:** Converts input tokens to dense vectors.
- **attention:** Dynamically focuses on encoder output at each step.
- **LSTMCell:** Processes current token + visual context.
- **log_softmax:** Converts decoder output to log-probabilities.
- **NLLLoss:** Compares predicted log-probs with true token.
- **Teacher Forcing:** Randomly feeds true token instead of predicted one.
- **clip gradients:** Prevents exploding gradients for stable training.
- **optimizer.step:** Updates model weights using gradients.

Evaluate Function Overview

```python
def evaluate(encoder, decoder, loader):
    encoder.eval()
    decoder.eval()

    smoothie = SmoothingFunction().method4
    bleu_scores = []

    print("\nEvaluating model...")
    progress_bar = tqdm(enumerate(loader), total=len(loader), desc="Evaluating")

    with torch.no_grad():
        for idx, (images, formulas) in progress_bar:
            if images is None or formulas is None:
                continue

            images = images.to(device)

            # Get the actual formula (removing special tokens)
            actual_formula = []
            for i in range(1, formulas.size(1)):  # Skip <sos>
                token_idx = formulas[0, i].item()
                if token_idx == word_to_index["<eos>"]:
                    break
                if token_idx != word_to_index["<pad>"] and token_idx != word_to_index["<unk>"]:
                    actual_formula.append(index_to_word[token_idx])

            # Get the predicted formula
            predicted_formula = predict_formula(images, encoder, decoder)

            # Filter out any unknown tokens from the prediction
            predicted_formula = [token for token in predicted_formula if token != "<unk>"]

            # Calculate BLEU score
            if not predicted_formula:
                bleu = 0.0
            else:
                try:
                    bleu = sentence_bleu([actual_formula], predicted_formula, smoothing_function=smoothie)
                except Exception as e:
                    print(f"BLEU calculation error: {e}")
                    bleu = 0.0

            bleu_scores.append(bleu)

            # Update progress bar
            avg_bleu = sum(bleu_scores) / len(bleu_scores)
            progress_bar.set_postfix(bleu=f"{avg_bleu:.4f}")

            # Print a few examples
            if idx < 5 or idx % (len(loader) // 10) == 0:
                print(f"\nSample {idx+1}:")
                print(f"Actual:    {' '.join(actual_formula)}")
                print(f"Predicted: {' '.join(predicted_formula)}")
                print(f"BLEU Score: {bleu:.4f}")

    average_bleu = sum(bleu_scores) / len(bleu_scores) if bleu_scores else 0
    print(f"\nAverage BLEU score: {average_bleu:.4f}")
    return average_bleu
```

Purpose of `evaluate` Function:

- Assesses model performance on the test dataset.
- Uses trained encoder decoder to predict formulas.
- Compares predictions with ground truth formulas.
- Calculates BLEU score for each sample.
- Returns the average BLEU score across the test set.
- Prints examples to provide qualitative feedback.

Training Loop Overview

```python
# Training loop
best_bleu = 0.0
best_epoch = 0

for epoch in range(num_epochs):
    print(f"\nEpoch {epoch+1}/{num_epochs}")

    # Train for one epoch
    train_loss = train_epoch(encoder, decoder, train_loader, optimizer, criterion)
    print(f"Training loss: {train_loss:.4f}")

    # Evaluate on validation set
    bleu = evaluate(encoder, decoder, test_loader)

    # Learning rate scheduling
    scheduler.step(train_loss)

    # Save checkpoint if this is the best model so far
    if bleu > best_bleu:
        best_bleu = bleu
        best_epoch = epoch + 1

        # Save the models
        checkpoint = {
            'epoch': epoch,
            'encoder': encoder.state_dict(),
            'decoder': decoder.state_dict(),
            'optimizer': optimizer.state_dict(),
            'bleu': bleu,
            'word_to_index': word_to_index,
            'index_to_word': index_to_word
        }
        torch.save(checkpoint, f"best_model_epoch{epoch+1}_bleu{bleu:.4f}.pth")
        print(f"New best model saved with BLEU score: {bleu:.4f}")

    # Regular checkpoint
    checkpoint = {
        'epoch': epoch,
        'encoder': encoder.state_dict(),
        'decoder': decoder.state_dict(),
        'optimizer': optimizer.state_dict(),
        'bleu': bleu
    }
    torch.save(checkpoint, f"checkpoint_epoch{epoch+1}.pth")

print(f"Training complete. Best model was at epoch {best_epoch} with BLEU score {best_bleu:.4f}")
```

Training and Evaluation:

- Model trains over multiple epochs.
- Each epoch:
  - Trains using `train_epoch()`.
  - Evaluates with BLEU score
  - Adjusts learning rate via scheduler.

Checkpointing:

- Best BLEU model is saved with full state.

Final Output:

- Prints best epoch and BLEU score

Performance Metrics

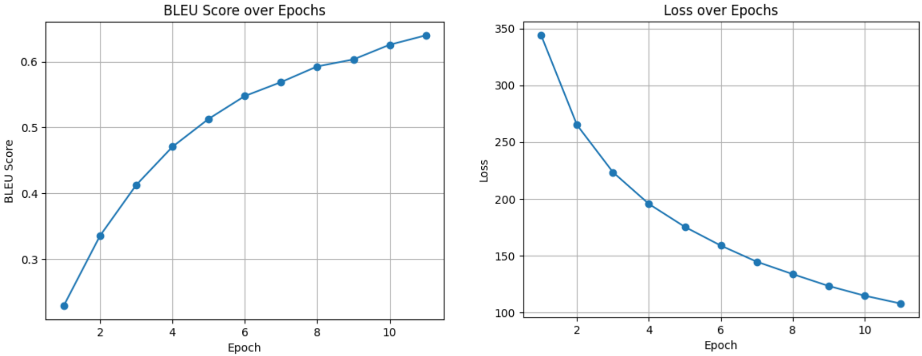
Loss Function (Negative Log-Likelihood - `NLLLoss`)

- Measures how well the predicted token probabilities match the target sequence.
- Computes the log loss between predicted probabilities and true tokens.
- Ignores padding tokens using `ignore_index = <pad>`.
- **Lower loss** indicates better performance during training.

BLEU Score (Bilingual Evaluation Understudy)

- Evaluates the quality of generated sequences against reference outputs.
- Based on n-gram precision (1-gram to 4-gram matches).
- Includes a *brevity penalty* to penalize overly short outputs.
- **Score range:** 0.0 (no match) to 1.0 (perfect match).
- Commonly used in translation and text generation tasks.

Results


BLEU Score over Epochs
BLEU Score
0.3
0.4
0.5
0.6
2
4
6
8
10
Epoch
Loss over Epochs
Loss
100
150
200
250
300
350
2
4
6
8
10
Epoch

Results

```
Sample 4451:
Actual:    $ b _ { \mu } \longrightarrow b _ { \mu } - s _ { \mu }
\; , $
Predicted: $ g _ { \mu } \rightarrow \delta _ { \mu } - B _
{ \mu } , $
BLEU Score: 0.4316

Sample 5341:
Actual:    $ \delta \varphi ^ { r s } = \left[ \eta _ { i } ^
{ \mu } \eta _ { j } ^ { \nu } \delta ^ { i r } \delta ^ { j s } -
( \eta _ { o } ^ { \mu } \eta _ { j } ^ { \nu } - \eta _ { o } ^
{ \nu } \eta _ { j } ^ { \mu } ) \frac { \delta ^ { i r } \delta ^
{ j s } p ^ { j } } { p ^ { o } + \sqrt { p ^ { 2 } } } \right]
\delta \omega _ { \mu \nu } . $
Predicted: $ \delta ^ { \mu } \varphi ^ { \mu } = \delta _ { i j } ^
{ a } \delta ^ { \mu } \delta ^ { \mu } - \partial _ { i } ^ { \mu }
- \partial _ { i } ^ { \mu } - \frac { \partial ^ { \mu } }
{ \partial { \bf p } } ^ { \mu } } { { \mu ^ { \mu } } } { \partial
{ { \mu } } { \mu } } } { { { { \mu }
BLEU Score: 0.2431

Sample 6231:
Actual:    $ I = I _ { + } + I _ { - } = \frac { 2 \pi } { \vert
\lambda \vert ^ { 2 } \vert a \vert ^ { 2 } } . $
Predicted: $ I = L _ { + } + \frac { 2 \pi } { 1 | | | | | | | | | ^
{ 2 } } . $
BLEU Score: 0.3888

Sample 7121:
Actual:    $ D _ { + } ( G ^ { - 1 } \dot { G } ) = 0 = \bar { D } _
{ + } ( G ^ { - 1 } \dot { G } ) , $
Predicted: $ D _ { + } ( G ^ { + } ) = 0 . $
BLEU Score: 0.1661

Sample 8011:
Actual:    $ { \cal C } = \sigma ^ { 2 } \otimes C \otimes \rho ^
{ 1 } $
Predicted: $ { \cal G } = \sigma ^ { 2 } \otimes \otimes \otimes
{ 0 } \otimes { 0 } $
BLEU Score: 0.4158

Sample 8901:
Actual:    $ V ( \vec { \varphi } ^ { 2 } ) = \frac { \mu ^ { 2 } }
{ 2 } \vec { \varphi } ^ { 2 } + \frac { \lambda } { 4 } ( \vec
{ \varphi } ^ { 2 } ) ^ { 2 } $
Predicted: $ V ( \vec { \varphi } ^ { 2 } ) = \frac { d ^ { 2 } }
{ 2 } \varphi ^ { 2 } + \frac { \lambda } { 4 } ( \varphi ^ { 2 } )
^ { 2 } $
BLEU Score: 0.7485
```

Figure: After Epoch 2

```
Sample 4451:
Actual:    $ b _ { \mu } \longrightarrow b _ { \mu } - s _ { \mu }
\; , $
Predicted: $ b _ { \mu } \rightarrow b _ { \mu } - s _ { \mu } \ , $
BLEU Score: 0.7417

Sample 5341:
Actual:    $ \delta \varphi ^ { r s } = \left[ \eta _ { i } ^
{ \mu } \eta _ { j } ^ { \nu } \delta ^ { i r } \delta ^ { j s } -
( \eta _ { o } ^ { \mu } \eta _ { j } ^ { \nu } - \eta _ { o } ^
{ \nu } \eta _ { j } ^ { \mu } ) \frac { \delta ^ { i r } \delta ^
{ j s } p ^ { j } } { p ^ { o } + \sqrt { p ^ { 2 } } } \right]
\delta \omega _ { \mu \nu } . $
Predicted: $ \delta \varphi ^ { n } = \left[ \eta _ { i } ^ { \mu }
\eta _ { j } ^ { i j } \delta ^ { j i } ( - \eta _ { j } ^ { \mu }
\eta _ { j } ^ { \nu } - \eta _ { j } ^ { \nu } \eta _ { j } ^
{ \nu } ) \frac { \delta ^ { n } \Phi ^ { i } } { p ^ { + } } }
\right] \delta _ { \mu
BLEU Score: 0.5734

Sample 6231:
Actual:    $ I = I _ { + } + I _ { - } = \frac { 2 \pi } { \vert
\lambda \vert ^ { 2 } \vert a \vert ^ { 2 } } . $
Predicted: $ I = I _ { + } + I _ { - } = \frac { 2 \pi } { | \lambda
| ^ { 2 } | a | ^ { 2 } } . $
BLEU Score: 0.7595

Sample 7121:
Actual:    $ D _ { + } ( G ^ { - 1 } \dot { G } ) = 0 = \bar { D } _
{ + } ( G ^ { - 1 } \dot { G } ) , $
Predicted: $ D _ { + } ( G ^ { - 1 } G G ) = 0 , \bar { D } _ { + }
( G ^ { - 1 } G ^ { i } ) , $
BLEU Score: 0.7054

Sample 8011:
Actual:    $ { \cal C } = \sigma ^ { 2 } \otimes C \otimes \rho ^
{ 1 } $
Predicted: $ { \cal C } = \sigma ^ { 2 } \otimes C \otimes \rho ^
{ 1 } $
BLEU Score: 1.0000

Sample 8901:
Actual:    $ V ( \vec { \varphi } ^ { 2 } ) = \frac { \mu ^ { 2 } }
{ 2 } \vec { \varphi } ^ { 2 } + \frac { \lambda } { 4 } ( \vec
{ \varphi } ^ { 2 } ) ^ { 2 } $
Predicted: $ V ( \vec { \varphi } ^ { 2 } ) = \frac { \mu ^ { 2 } }
{ 2 } \dot { \varphi } ^ { 2 } + \frac { \lambda } { 4 } ( \varphi ^
{ 2 } ) ^ { 2 } $
BLEU Score: 0.8526
```

Figure: After Epoch 10

References

Aurélien Géron,
Hands-on Machine Learning with Scikit-Learn, Keras and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems,
O'Reilly Media, 2nd edition, 2019, ISBN 978-1492032649.

Charu C. Aggarwal,
Neural Networks and Deep Learning: A Textbook,
Springer, 1st edition, 2018, ISBN 978-3319944623.

Fei-Fei Li, Justin Johnson, and Serena Yeung,
CS231n: Convolutional Neural Networks for Visual Recognition,
Available online: `https://cs231n.github.io/`, Accessed: 2025-04-01.

Thank you