Generation model:

**1. Model Architecture**

**GAN Framework for Captcha Generation**

* **Generator (Residual CNN + PixelShuffle):**
  + **Purpose:** To generate high-quality, robust captcha images conditioned on a latent noise vector and (optionally) text input.
  + **Architecture Highlights:**
    - Incorporates a text embedding module (if conditional generation is required) that projects one-hot encoded captcha text to an 80-dimensional space.
    - Concatenates the latent noise vector (default dimension 100) with the text embedding to produce a flattened feature vector.
    - Uses a fully connected layer to produce a feature map (reshaped to [batch, 512, 4, 8]).
    - Applies several upsampling blocks, each consisting of:
      * A convolution that temporarily expands channels (by a factor, e.g. 4),
      * A PixelShuffle operation (with upscale factor = 2) that increases spatial resolution, and
      * A “post” convolution layer to refine the features with batch normalization and ReLU activation.
    - Two residual blocks (each with two 3×3 convolutions, batch normalization, ReLU, and a skip connection) are used at the end.
    - Outputs the final image using a last convolution layer followed by a *tanh* activation.
* **Discriminator (Spectral Normalized CNN):**
  + **Purpose:** To distinguish real captcha images from generated ones.
  + **Architecture Highlights:**
    - Uses a series of spectral normalized convolutional layers, increasing channels in the order: 3 → 32 → 64 → 128 → 256.
    - Each convolution uses a kernel size of 4, stride 2, padding 1, and is followed by LeakyReLU activation.
    - Dropout (0.3) is applied to reduce overfitting.
    - The final classifier flattens the features and uses a spectral normalized fully connected layer to output a single realism score.
* **Solver (CNN+LSTM):**
  + **Purpose:**  
    To “read” the generated captcha images by predicting their textual content. The solver is implemented as a combination of a CNN (for feature extraction) and an LSTM (for sequential decoding).
  + **Architecture Highlights:**
    - The network outputs log probabilities over character classes and is trained using a CTC (Connectionist Temporal Classification) loss.
    - It serves as an adversarial component during training by enforcing that generated captchas are difficult for the solver to decode correctly.

**2. Parameters**

* **General Parameters:**
  + **Batch Size:** 64
  + **Latent Dimension (Generator input):** 100
  + **Pretraining Epochs:** e.g., 5 (for initial conditional training with L1 loss)
  + **Total Training Epochs:** 100
* **Learning Rates:**
  + Generator: e.g., 0.0002
  + Discriminator: e.g., 0.0001
  + Solver: e.g., 0.001
* **Regularization & Loss Weighting:**
  + Gradient Penalty Lambda (λ\_gp): 10
  + L1 Loss Weight during pretraining (λ\_l1): 100
  + Solver Loss Weight (λ\_solver): 1.0
* **Text/Label Specific:**
  + The captcha is generated for a fixed label length (e.g., 5) using a character set (combining digits, uppercase, and lowercase letters).
  + One-hot encoding is used to represent text inputs, which are then embedded and concatenated with the noise vector.

**3. Loss Functions & Metrics**

**Discriminator Loss**

* **Wasserstein Loss with Gradient Penalty:**

lossD=E[D(fake)]−E[D(real)]+λgp×GP\text{loss}\_D = \mathbb{E}[D(\text{fake})] - \mathbb{E}[D(\text{real})] + \lambda\_{gp} \times \text{GP}

* + The gradient penalty (GP) is computed on interpolated samples to enforce the Lipschitz constraint.
* **Optional FGSM Component:**
  + When enabled (controlled by a flag), FGSM (Fast Gradient Sign Method) is applied to both real and fake images before feeding them to D.
  + Perturbations are computed and added (with an epsilon factor that is adaptively modified), and the discriminator loss incorporates both the original and perturbed image losses (averaged).

**Generator Loss**

* **Adversarial Loss:**

lossG\_adv=−E[D(gen)]\text{loss}\_{G\\_adv} = -\mathbb{E}[D(\text{gen})]

* **L1 Loss (During Pretraining):**

lossG\_L1=L1Loss(gen\_images,real\_images)×λl1\text{loss}\_{G\\_L1} = \text{L1Loss}(\text{gen\\_images}, \text{real\\_images}) \times \lambda\_{l1}

* **Solver-Based Loss:**
  + The solver network provides an additional loss by measuring its ability to decode the generated images using a CTC loss. Its negative value is added to the generator loss, weighted by λ\_solver.
* **Total Generator Loss:**
  + During pretraining, the L1 loss is used to make the generator output similar to real images.
  + Later, the total generator loss combines adversarial loss and the solver-based loss.

**Solver Loss**

* **CTC Loss:**  
  The solver predicts text from images and its performance is measured using a CTC loss function (which accounts for alignment without needing explicit segmentation).
* **FGSM Perturbation for Solver:**
  + The solver loss is computed on both clean and FGSM-perturbed images.
  + The final solver loss is the average of the losses on clean and adversarial images.

**4. Training Method**

**Data Preparation**

* **Dataset:**  
  Custom datasets (either CaptchaDataset or FGSM\_dataset) load captcha images and text labels using one-hot encodings.
* **DataLoader:**  
  The DataLoader shuffles the data and maintains the batch size of 64.

**Alternating Optimization**

The training loop alternates the updates among the discriminator, solver, and generator.

**Discriminator Update**

* **Standard Update:**
  + Freeze the generator and solver.
  + Feed both real images and fake images (drawn from a historical image buffer for stability) into the discriminator.
  + Compute the standard Wasserstein loss and add the gradient penalty.
* **Optional FGSM-Based Enhancement:**
  + If enabled, FGSM is applied to both real and fake images to create adversarial examples.
  + The discriminator computes scores for both clean and perturbed images.
  + The final loss is averaged between the original and FGSM-perturbed losses, then combined with the gradient penalty.

**Solver Update (CNN+LSTM)**

* **Primary Goal:**
  + Train the solver to correctly read text from captcha images.
* **Process:**
  + Compute the CTC loss on real images.
  + Generate adversarial images using FGSM (based on the gradients from the CTC loss), then compute the CTC loss on these perturbed images.
  + The overall solver loss is the average of clean and adversarial losses.
  + Additionally, the solver is updated on fake images from the generator to reinforce robust text recognition.

**Generator Update**

* **Objective:**
  + Train the generator to both fool the discriminator and produce captchas that are challenging for the solver.
* **Process:**
  + Freeze the discriminator and solver.
  + Compute the adversarial loss from the discriminator.
  + Use the solver’s CTC loss (with its negative sign) as an additional term, penalizing the generator if the solver can easily decode the generated text.
  + During pretraining (for a number of epochs), an L1 loss is used to bring generated images closer to real images.
  + The generator is updated several times per discriminator update for stability.

**Optional FGSM Component Details**

* **For Discriminator:**
  + If the FGSM flag is set, gradients are computed for real and fake images, and a perturbation (scaled by an epsilon factor) is added before re-evaluating the discriminator.
  + The loss becomes an average of losses on clean and adversarial images.
* **For Solver:**
  + FGSM is similarly applied to generate adversarial images from real ones.
  + The solver minimizes the average CTC loss across both sets.
* **Adaptive Epsilon Adjustment:**
  + The epsilon factor, which controls the perturbation strength, is adjusted based on the solver’s accuracy (increasing if solver accuracy >70%, decreasing if below 30%).

**Optimizers**

* Separate Adam optimizers are employed for the generator, discriminator, and solver with their respective learning rates and beta parameters.

**Summary of Purpose and Training Considerations**

* **Purpose:**
  + The GAN is used to generate captchas that are robust against solver attacks. This branch task can also be used for data augmentation by generating additional captcha images to train solvers.
  + The later inclusion of FGSM provides an adversarial training aspect, enhancing both the solver and generator so that the generated captchas are increasingly challenging.
* **Model Structure:**
  + **GAN:**
    - Generator: Residual CNN + PixelShuffle
    - Discriminator: Spectral Normalized CNN
  + **Solver:**
    - CNN+LSTM-based network
* **Operational Note:**
  + Since deviating from our primary purpose (captcha generation and robustness) requires long-term training, the primary goal here is to ensure that the model runs correctly. No further extensive training is provided in this code.

**5. Evaluation Results**

* **Qualitative Evaluation:**
  + At the end of each epoch, the generator is set to evaluation mode, and a set of sample images is generated using random noise and condition text (sampled from the character set).
  + These generated images are saved in a result directory, with filenames that document the generated text.
* **Solver Accuracy Evaluation:**
  + The solver’s output is decoded (via a custom decoding function that collapses repeated and blank predictions) and compared against the target strings to compute an accuracy metric.
  + The solver’s accuracy is used to adjust the FGSM epsilon dynamically during training.
* **Loss Metrics:**
  + Training statistics such as discriminator loss, generator adversarial loss, and solver accuracy (plus current epsilon) are printed at the end of each epoch.
  + These metrics help monitor the stability and progress of training.

**1. Model Architecture**

**Recognition Model: SAR**

This model adopts the basic design of SAR (Segmentation Aware Recognition) with the following main flow:

* **CNN Feature Extractor**  
  Multiple convolutional layers (with BatchNorm and ReLU) are used to extract local features from the input captcha image. An adaptive average pooling layer is then applied to compress the height of the feature maps—while retaining the width information—to create a sequential feature representation suitable for later sequence modeling.
* **Bi-directional LSTM Encoder**  
  The CNN-extracted features are fed into a bi-directional LSTM to perform global sequential modeling.
  + With the bi-directional structure, the output feature dimension at each timestep is twice the hidden size (e.g., with a hidden size of 256, the output will be 512).
  + A dropout (0.5) is applied internally to reduce overfitting.
* **Attention Block Decoder**  
  An attention mechanism combined with an LSTMCell is used to generate the character sequence step by step:
  + **Input Preparation:** An embedding layer converts character indices into vector representations, and the initial input is the <sos> (start-of-sequence) token.
  + **Step-wise Decoding:** At each step, the input consists of the embedding of the previously generated character and the current context vector, and the LSTMCell updates the hidden and cell states.
  + **Attention Module:** The encoder outputs are linearly transformed and then matched with the current hidden state to calculate attention scores. A softmax is applied to obtain attention weights, which are used to compute a dynamic context vector by weighted summation.
  + **Output Generation:** The decoder's hidden state (after dropout) is concatenated with the context vector and passed through a fully connected layer to produce logits for each character class.
  + **State Initialization:** The initial hidden and cell states for the decoder are generated by mapping the mean of the encoder outputs through linear layers.

Overall, the model follows the flow:  
**CNN-extract feature → Bi-LSTM Encoder → Attention block decoder**

**2. Parameters**

1. **Character Dictionary and Special Tokens:**
   * The basic character set consists of digits (0–9) and lowercase letters (a–z).
   * Special tokens include <pad> (padding), <sos> (start-of-sequence), and <eos> (end-of-sequence).
   * Bidirectional mappings (char2idx and idx2char) are constructed to encode and decode labels.
2. **CNN Module Parameters:**
   * **First Layer:** Input channels 3 → Output channels 64; kernel\_size=3, stride=2, padding=1
   * **Second Layer:** 64 → 128; kernel\_size=3, stride=2, padding=1
   * **Third Layer:** 128 → 256; kernel\_size=3, stride=(2,1), padding=1
   * **Fourth Layer:** 256 → 256; kernel\_size=3, stride=(2,1), padding=1
3. **EncoderRNN Parameters:**
   * Hidden unit size: 256
   * The bi-directional LSTM produces an output with dimension 2 × 256 = 512
   * Dropout rate is set to 0.5
4. **AttentionDecoder Parameters:**
   * Embedding dimension: 128
   * Decoder hidden unit size: 256
   * Attention projection dimension: 128
   * Vocabulary size is determined by the size of the character set
   * A dropout (0.5) is applied internally, and linear layers are used to initialize the decoder states from the mean of the encoder outputs.
5. **Training Hyperparameters:**
   * **Optimizer:** Adam with an initial learning rate of 0.001 and a weight decay of 1e-5
   * **Learning Rate Scheduler:** ReduceLROnPlateau (with a patience of 3 and a factor of 0.5) to adjust the learning rate based on the validation loss
   * **Batch Size:** 32
   * **Data Preprocessing:**
     + Training data undergoes data augmentation (random rotation, color jitter, affine transformation, and uniform resizing to (79,729) followed by normalization).
     + Test data is only resized and normalized.

**3. Loss Function and Match Rate Metrics**

1. **Loss Function:**
   * The cross-entropy loss (nn.CrossEntropyLoss) is used, with the parameter ignore\_index=char2idx[pad\_token] to ignore the padded tokens.
   * The loss is computed by comparing the logits produced by the decoder (for each timestep) with the corresponding true character labels.
2. **Match Rate Metrics:**
   * **Single Character Match Rate:**  
     The accuracy of matching each character at its corresponding position between the predicted string and the true label. This measures local prediction accuracy.
   * **Complete Word Match Rate:**  
     Only when the entire predicted captcha string exactly matches the true string is it considered a match, reflecting the overall generation accuracy.
   * **Captcha Match Rate:**  
     This metric is based on the Longest Common Subsequence (LCS) between the predicted string and the true label—calculating the ratio between the length of the LCS and the length of the true label. This metric provides a comprehensive evaluation of the overall matching quality.

Different strategies (e.g., employing Beam Search during decoding) affect these match rate metrics, which serve as crucial indicators for evaluating model performance.

**4. Training Method**

1. **Data Loading and Preprocessing:**
   * A custom dataset (CaptchaDataset) is used to read images and extract the true captcha string from file names, appending the <eos> token at the end of each label.
   * Data is split using random\_split into 90% for training and 10% for validation, with a separate test set built independently.
   * A custom pad\_collate\_fn is employed to pad variable-length label sequences (with <pad> tokens) for batching.
2. **Training Process:**
   * **Forward Pass:**  
     Images pass through the CNN for feature extraction, then into the bi-directional LSTM encoder, and finally into the attention decoder to generate the character sequence step-by-step.
   * **Loss Calculation and Backward Pass:**  
     The loss is computed using cross-entropy (with <pad> tokens ignored), and the model weights are updated via the Adam optimizer.
   * **Learning Rate Scheduling:**  
     The ReduceLROnPlateau scheduler dynamically adjusts the learning rate based on the validation loss.
   * **Regularization:**  
     Dropout (0.5) is applied in both the encoder and decoder to mitigate overfitting.
3. **Training Enhancement Strategies and Post-processing:**
   * **Baseline Approach:** The basic training setup yields initial performance results.
   * **Enhancements Using Scheduler, Regularization, and Dropout:** Combining data augmentation with a dynamically adjusted learning rate and dropout significantly improves model performance.
   * **Beam Search Decoding:** Incorporating Beam Search during inference further improves the complete word match rate and captcha match rate.

**5. Evaluation Results**

Under various training and post-processing strategies, the specific match rate metrics are as follows:

* **Results after applying Scheduler-adjusted lr, regularization, dropout, and data augmentation:**
  + Single Character Match Rate: **0.8672**
  + Complete Word Match Rate: **0.5506**
  + Captcha Match Rate: **0.8810**
* **Results using Beam Search during decoding:**
  + Single Character Match Rate: **0.8642**
  + Complete Word Match Rate: **0.5521**
  + Captcha Match Rate: **0.8757**

*Metric Explanations:*

* **Single Character Match Rate:** Measures the accuracy of each character's matching at corresponding positions.
* **Complete Word Match Rate:** Only counts as a match when the entire predicted captcha is exactly the same as the true label.
* **Captcha Match Rate:** Calculated based on the LCS, reflecting overall matching performance; this metric may be further refined or extended.

