New York University Tandon School of Engineering
Computer Science and Engineering

CS-GY 6923: Optional Project.
Due Friday, May 1st, 2026, 11:59pm.

Discussion with other students and AI models is allowed for this project, but solutions must be written-up individually.

Overview

In this project, you will explore scaling laws for language models trained on SVG (Scalable Vector Graphics) code. SVG is a structured, non-linguistic domain: it is XML-based text that encodes visual content such as icons, emoji, and shapes. Unlike natural language, SVG has strict syntactic rules (valid XML, coordinate systems, color values), hierarchical structure (nested groups, transforms), and the outputs can be **instantly rendered and visually inspected** in any browser — making evaluation both quantitative and qualitative.

You will train decoder-only **Transformer** language models at multiple scales, fit power-law scaling curves, investigate how learning rate tuning strategies affect scaling behavior, and generate and evaluate SVG samples.

Deliverables:

- PDF report (6–10 pages recommended, excluding references and appendices)
- Code repository with all scripts and a README

Grading Policy: This is an optional extra credit project. Partial credit is available for incomplete but well-executed work. You will not be penalized for not doing this project: we will assign initial grades based on the labs, homeworks, midterms and finals, ignoring the project; we will only update the grades for the students who submit the project. Only submit your report if you actually do substantial work on the project, we will not give you any points for a half-baked report you produce in a couple hours. If you submit this project, you should be prepared to answer technical questions about your work. If you cannot answer reasonable questions about your methods, results, or code, you will receive **negative** points for the project. Presenting fake results in your report (i.e. just making up experimental results) is equivalent to cheating and will have consequences.

Project Goals

By the end of this project, you should:

1. Build a complete data preprocessing pipeline for SVG code
2. Empirically derive scaling laws for transformer-based language models on SVG data
3. Investigate how learning rate tuning strategies (fixed vs. μP transfer) affect scaling behavior
4. Analyze what visual and structural patterns emerge at different model scales
5. Generate and evaluate SVG samples from your best model

Part 1: Data Collection and Preprocessing (15%)

Recommended Datasets

Primary Dataset: **starvector/svg-icons-simple** (HuggingFace)

- ∼89,370 simplified SVG icons, 153 MB total
- Already preprocessed/simplified SVG code (short, clean)
- From the StarVector project (Rodriguez et al., 2023)
- Load with: `load_dataset("starvector/svg-icons-simple")`
- URL: https://huggingface.co/datasets/starvector/svg-icons-simple

Supplementary Datasets (to reach token count targets):

- **starvector/svg-emoji-simple**: Simplified SVG emoji, 14.5 MB (https://huggingface.co/datasets/starvector/svg-emoji-simple)
- **starvector/svg-fonts-simple**: Simplified SVG font glyphs, 2.38 GB — subsample as needed (https://huggingface.co/datasets/starvector/svg-fonts-simple)
- **starvector/svg-stack-simple**: Larger/more diverse simplified SVGs, 3.87 GB (https://huggingface.co/datasets/starvector/svg-stack-simple)
- **umuthopeyildirim/svgen-500k**: ∼300K SVGs with name/label, SVG code, and text descriptions (https://huggingface.co/datasets/umuthopeyildirim/svgen-500k)

Use **svg-icons-simple** as the primary dataset. If more data is needed to hit the 100M token minimum, combine with svg-emoji-simple and/or subsample from svg-fonts-simple. You may also use any other SVG datasets you like, as long as you clearly describe the source in your report and use the data systematically. Filter out SVGs exceeding a maximum token length (suggested: 1024 or 2048 tokens) to keep context window requirements manageable for smaller models.

Your Tasks

1. **Download and prepare data** from the HuggingFace datasets listed above.
2. **SVG normalization/cleaning:**
   - Strip comments, metadata, and unnecessary whitespace
   - Normalize coordinate precision (e.g., round to 1 decimal place) to reduce vocabulary
   - Optionally canonicalize attribute ordering
   - Remove SVGs that are too short (< 50 characters) or too long (above your chosen token threshold)
   - Validate that all SVGs in the cleaned set parse as valid XML
   - Ensure all SVGs render without errors (use `lxml` for XML parsing, optionally `CairoSVG` for render validation)
3. **Tokenization:** Train a BPE tokenizer on your SVG corpus using `sentencepiece` or the HuggingFace `tokenizers` library. Vocabulary sizes in the range 1K–8K are reasonable. Document the vocabulary size and justify your choice in the report.
4. **Create train/validation/test splits:**
   - Recommended: 98% / 1% / 1% (by number of SVG files, before concatenation)
   - Ensure at least 100M tokens in the training set
   - Split by file (not by token position within concatenated text) to avoid data leakage
5. **Document dataset statistics:**
   - Vocabulary size
   - Total token count (train/val/test)

- Sequence length distribution (histogram)
- Number of files before/after filtering
- Examples of SVGs at different complexity levels (rendered)

Deliverable: Clear description of the preprocessing pipeline with statistics. Include rendered examples of SVGs from the dataset at various complexity levels.

Part 2: Transformer Scaling Study (35%)

Train a family of decoder-only transformer language models of varying sizes on the SVG data. Measure the validation loss after **1 epoch** of training for each model size.

Model Sizes to Test

Train at least **5 different model sizes**. Suggested configurations (adjust based on your computational resources):

Name	∼Params	d_model	n_layers	n_heads	d_ff
Tiny	∼1M	128	4	4	512
Small	∼3M	192	6	6	768
Medium	∼10M	384	6	6	1536
Large	∼30M	512	10	8	2048
XL	∼88M	768	12	12	3072

Vary `n_layers`, `n_heads`, `d_model`, `d_ff` to achieve different parameter counts.

Compute Planning: Plan carefully for the compute available to you and the size of the experiments you intend to run. Before launching large training runs, estimate the time and memory requirements based on your smaller runs. If your resources are limited, you may scale down the largest model sizes, but you must provide clear reasoning and justify this decision in your report.

Requirements

1. **Learning rate sweep on the smallest model:**
   - Perform a learning rate sweep on your **smallest** model (e.g., Tiny). Test at least 5–7 learning rates on a log scale.
   - Use a cosine learning rate schedule with warmup for each run.
   - Select the best learning rate based on validation loss.
   - Use this same learning rate for all larger model sizes in this part.
2. **Use consistent training setup** across all models:
   - Same tokenization scheme
   - Same learning rate schedule (cosine with warmup), with the learning rate selected above
   - Same batch size (measured in tokens)
   - Same training data (at least 100M tokens)
   - Train for exactly 1 epoch for the scaling plot comparison
   - Optimizer: AdamW is recommended
3. **Create a scaling plot:**
   - X-axis: Number of parameters (log scale)

- Y-axis: Validation loss after 1 epoch
- Fit a power law: $L = a \cdot N^{-\alpha} + c$, where N is the parameter count
- Report the fitted scaling exponent α and discuss its implications

4. **Track additional metrics:**
   - Training loss curves over time for each model
   - Wall-clock time per epoch
   - GPU memory usage
   - Tokens per second throughput

Reference Code

You may use **nanoGPT** (https://github.com/karpathy/nanoGPT) as a starting point. Clearly document in your report what you borrowed versus what you modified or implemented yourself.

Deliverable: Learning rate sweep results for the smallest model, scaling plot with power law fit, training curves for all models, table of model architectures and training statistics, and analysis of how loss decreases with model size.

Part 3: μP Scaling and Extrapolation (25%)

In Part 2, you tuned the learning rate on the smallest model and used it for all model sizes. In this part, you will investigate whether **μP (Maximal Update Parameterization)** can improve scaling behavior by enabling principled learning rate transfer across model widths. You will also use your scaling laws to make predictions beyond the model sizes you trained.

Background: μP

A key practical challenge in scaling up neural networks is hyperparameter tuning: the optimal learning rate for a small model is generally not optimal for a larger model under standard parameterization (SP). **μP** is a principled reparameterization of neural network layers that enables **zero-shot hyperparameter transfer** across model widths. Under μP, the optimal learning rate found for a small "proxy" model transfers directly to wider models without retuning.

Key reference: Yang et al. (2022), "Tensor Programs V: Tuning Large Neural Networks via Zero-Shot Hyperparameter Transfer" (https://arxiv.org/abs/2203.09789)

Implementation: Use the `mup` Python package (https://github.com/microsoft/mup). The package provides drop-in replacements for standard PyTorch layers that implement μP scaling rules. The key changes are to the initialization scale and per-layer learning rate multipliers as a function of model width. **Note:** applying μP to transformers requires specific modifications, including changing the attention scaling from $1/\sqrt{d}$ to $1/d$ and careful initialization of certain layers. See the "μP for Transformers" section of the `mup` README for details.

μP Scaling Study

1. **Reparameterize your model using μP** (via the `mup` package).
2. **Perform a learning rate sweep** on the smallest μP model (same sweep protocol as Part 2: at least 5–7 learning rates on a log scale).
3. **Transfer the optimal learning rate** to all larger μP models *without retuning*.
4. **Train all model sizes** for 1 epoch (same data, tokenization, and batch size as Part 2).
5. **Compare standard parameterization (Part 2) vs. μP:**
   - Plot both scaling curves (fixed LR from Part 2 and μP) on the same graph

- Does μP improve validation loss, especially for larger models?
- Fit power laws to both curves. Does μP yield a better (steeper) scaling exponent?
- Show the learning rate sweep results for the small model under both parameterizations
- Discuss: Why might a fixed learning rate degrade for larger models? How does μP address this?

Scaling Law Extrapolation

Using your best scaling law fit from Parts 2–3 (whichever approach gave the better power law):

1. **Predict** the validation loss you would expect from a model with **10× more parameters** than your largest (XL) model. Report the predicted loss along with a confidence interval or uncertainty estimate from your fit.
2. **Discuss:** How confident are you in this prediction? What factors might cause the actual loss to deviate from the extrapolation? How far beyond your data do you think the power law remains reliable?

Deliverable: μP learning rate sweep results, comparison plot of standard vs. μP scaling curves with power law fits, extrapolation prediction with discussion of uncertainty, and analysis of when and why μP helps.

Part 4: Best Model Training and Sample Generation (15%)

Training

1. Choose your best model (likely the largest feasible transformer with the best LR strategy)
2. Train for as many tokens/epochs as feasible
3. Tune hyperparameters if time permits (learning rate, dropout, weight decay, etc.)

Generation

Generate samples from your trained model:

- At least 10 **unconditional** samples (generate from an `<svg` prefix)
- At least 5 **prefix-conditioned** samples (provide partial SVGs and let the model complete them). Examples of interesting prefixes:
  - A partial face (circle + one eye) — does the model add the other eye and mouth?
  - An open path — does the model close it?
  - A group with one shape — does the model add related shapes?
- Use temperature sampling with different temperatures (e.g., 0.5, 0.8, 1.0) and compare
- Use top-k and/or nucleus (top-p) sampling

Evaluation

Quantitative Metrics:

- Final perplexity on test set
- **XML validity rate:** percentage of generated outputs that parse as valid XML (use `lxml.etree`)
- **SVG render rate:** percentage that successfully render to PNG (use CairoSVG: `pip install cairosvg`)
- **Structural validity:** percentage with correct `<svg>` root element, properly closed tags, valid attribute values

Qualitative Analysis:

- Do generated SVGs look like coherent icons/shapes?
- Does the model learn spatial structure (e.g., symmetry, centering)?
- Does the model learn SVG conventions (e.g., viewBox usage, common color palettes)?
- What patterns emerge at different sampling temperatures?
- Include a rendered grid of generated samples in the report

Prefix Completion Analysis:

- Show the prefix, the model's completion, and the rendered result side by side
- Does the model produce contextually appropriate completions?

Deliverable: Generated samples (both SVG code and rendered images), quantitative evaluation metrics, qualitative analysis with visual examples, discussion of prefix completion behavior.

Part 5: Design Decisions and Analysis (10%)

Throughout your report, clearly document and justify all major decisions:

1. **Tokenization strategy** — Why did you choose your approach? What vocabulary size and why? How does tokenization affect sequence length and model performance?
2. **Architecture choices** — Model sizes, layer configurations, context window length.
3. **Training decisions** — Batch size, learning rate schedule, optimizer choice, sequence length, regularization.
4. **Scaling insights** — How do your results compare to known scaling laws (Kaplan et al., 2020; Hoffmann et al., 2022)? Is the scaling exponent for SVG similar to or different from natural language? What does this tell us about the structure of the SVG domain?
5. **Learning rate scaling insights** — How did fixed LR vs. μP compare? At what model size did the difference become significant? What does this tell you about hyperparameter transfer in practice?
6. **SVG-specific patterns** — What visual/structural concepts did models learn at different scales? Did smaller models learn basic syntax while larger models learned spatial coherence? Were there apparent phase transitions?
7. **Challenges encountered** — What didn't work? What would you do differently with more time or resources?

Deliverables

Code Repository

Your code submission should include:

- All preprocessing scripts (data download, cleaning, tokenization)
- Transformer training code (can be based on nanoGPT)
- μP implementation and learning rate sweep scripts
- Evaluation scripts (validity checking, rendering, metrics)

- Sample generation scripts
- Configuration files for all model architectures
- `README.md` with setup and usage instructions
- `requirements.txt` with all dependencies

PDF Report

Your report should be 6–10 pages (excluding references and appendices) and include:

Introduction (0.5–1 page)

- Motivation for studying scaling laws on SVG data
- Overview of your approach

Data (1–1.5 pages)

- Dataset description and preprocessing pipeline
- Tokenization scheme with examples
- Statistics and rendered SVG examples

Methods (1.5–2 pages)

- Model architectures (table of configurations)
- Training setup and hyperparameters
- Experimental design for scaling studies
- Evaluation metrics

Results (2–3 pages)

- Transformer scaling plot with power law fit
- Fixed LR vs. μP comparison and learning rate sweep results
- Scaling law extrapolation prediction
- Training curves
- Generated samples with evaluation

Discussion (1–2 pages)

- Key insights from scaling experiments
- Interpretation of learning rate scaling and μP results
- Design decisions and their impact
- Limitations and future work

Conclusion (0.5 page)

- Summary of main findings

Appendices (optional, not counted toward page limit)

- Additional generated SVGs (rendered)
- Extended experimental details
- Code snippets for key components

Additional Resources

Code and Libraries:

- nanoGPT: https://github.com/karpathy/nanoGPT
- μP: https://github.com/microsoft/mup
- PyTorch: https://pytorch.org
- CairoSVG (for rendering): https://cairosvg.org
- lxml (for XML validation): https://lxml.de

Background Reading:

- Kaplan et al. (2020): "Scaling Laws for Neural Language Models" — https://arxiv.org/abs/2001.08361
- Hoffmann et al. (2022): "Training Compute-Optimal Large Language Models" (Chinchilla) — https://arxiv.org/abs/2203.15556
- Yang et al. (2022): "Tensor Programs V: Tuning Large Neural Networks via Zero-Shot Hyperparameter Transfer" (μP) — https://arxiv.org/abs/2203.09789
- Rodriguez et al. (2023): "StarVector: Generating Scalable Vector Graphics Code from Images and Text" — https://arxiv.org/abs/2312.11556

Related Work on SVG Generation with LLMs:

- Xing et al. (CVPR 2025): "Empowering LLMs to Understand and Generate Complex Vector Graphics" — https://arxiv.org/abs/2412.11102
- UniSVG (2025): "A Unified Dataset for Vector Graphic Understanding and Generation" — https://arxiv.org/abs/2508.07766
- Jain et al. (2023): "VectorFusion: Text-to-SVG by Abstracting Pixel-Based Diffusion Models" — https://arxiv.org/abs/2211.11319