HOMEWORK 0
PYTORCH PRIMER *

10-423/10-623 GENERATIVE AI
http://423.mlcourse.org

OUT: Aug. 27, 2025
DUE: Sep. 8, 2025
TAs: Abishek, Aryaman, Natalie, Rithvik, Somil

Instructions

- **Collaboration Policy**: Please read the collaboration policy in the syllabus.
- **Late Submission Policy:** See the late submission policy in the syllabus.
- **Submitting your work:** You will use Gradescope to submit answers to all questions and code.
  - **Written:** You will submit your completed homework as a PDF to Gradescope. Please use the provided template. Submissions can be handwritten, but must be clearly legible; otherwise, you will not be awarded marks. Alternatively, submissions can be written in LaTeX. Each answer should be within the box provided. If you do not follow the template, your assignment may not be graded correctly by our AI assisted grader and there will be a **2% penalty** (e.g., if the homework is out of 100 points, 2 points will be deducted from your final score).
  - **Programming:** You will submit your code for programming questions to Gradescope. There is no autograder. We will examine your code by hand and may award marks for its submission.
- **Materials:** The data that you will need in order to complete this assignment is posted along with the writeup and template on the course website.

Question	Points
LaTeX Template Alignment	0
Background Reading	6
Image Classification	29
Text Classification	19
Code Upload	0
Collaboration Questions	2
Total:	56

*Compiled on Monday 8th September, 2025 at 04:14

Introduction

In this assignment, you will choose-your-own-adventure as you get up to speed on (or do a quick review of) PyTorch and Weights & Biases.[1]

PyTorch is a general purpose deep learning library. It allows you to define a computation graph, `loss`, dynamically in Python, and then a simple call to `loss.backward()` computes all the adjoints (aka. gradients of the loss with respect to each parameter) for you. Gone are the days in which we needed to work through complicated matrix calculus just to train our models. Well of course, you'd very much need to do that for any function that isn't easily or efficiently expressed in PyTorch, but those cases are becoming less and less common.

Weights & Biases is a logging tool that allows you to easily track the behavior of your model during training and evaluation. As well, once you've logged the interesting bits of data (say, the validation loss every 10 epochs), you can easily create a plot with just a few clicks showing that information. There is another advantage: Each run of your code might be with different hyperparameters and, if you carefully log these as well, then with a few more clicks you can compare the model's behavior across different hyperparameter settings.

At a high-level, you will proceed as follows:

1. Read the PyTorch tutorial.
2. Read the Weights and Biases (wandb) tutorial.
3. Review the HW0 starter code. You'll find it closely mirrors the code described in the PyTorch tutorial.
4. Modify the starter code so that it incorporates Weights & Biases logging.
5. Run the requested experiments and report your results as tables/plots from the wandb interface.
6. Modify your code further so that it supports a different model (you will choose the model!).
7. Allow your code to choose a different optimizer (you will choose the optimizer!).
8. Run additional experiments in order to better understand PyTorch.

You will carry out these tasks on two applications: image classification and text classification.

[1] Although all students in this class have taken an Introduction to Machine Learning course before, some of those (even here at CMU) did cover PyTorch and others did not. We want to ensure that everyone here is ready to start HW1 at the same level.

Computing Environment

First you need to setup your computing environment. Below we outline how you could do so on your laptop, or on Google Colab.

Local Environment

To use PyTorch on your laptop, we recommend the following setup.

1. Follow the instructions linked below to install Python using MiniConda.

   https://docs.conda.io/projects/miniconda/en/latest/miniconda-install.html

   Then create and activate a new python environment. For example:

   ```
   conda create -n py312 python=3.12
   conda activate py312
   ```

2. Next follow the instructions for your machine to install PyTorch version 2.5.1 (the latest stable version).

   https://pytorch.org/get-started/locally/

3. Install Weights & Biases https://docs.wandb.ai/quickstart

   ```
   pip install wandb
   ```

4. Install Pandas

   ```
   pip install pandas
   ```

5. Install any other Python packages you may need with conda when possible, and pip otherwise.

   ```
   conda install <package>
   pip install <package>
   ```

Colab

Google Colab provides free easy access to some amount of GPU compute. On the free tier, your GPU jobs will time out after a fixed number of hours and you may be temporarily unable to use a GPU if you use too many hours. The limits are dynamic and not clearly documented.

There are two ways to use Colab:

1. **As a Jupyter Notebook:** To see an example of PyTorch in Colab, click the *Run in Google Colab* link from the tutorial: https://pytorch.org/tutorials/beginner/basics/intro.html

2. **As a Terminal:** You can also treat Google Colab as a VM and run code as you would at the terminal. You should first put your code and data in a Google Drive folder. Then create a Code cell with the following snippet and run it to mount your Google Drive folder.

   ```
   from google.colab import drive
   drive.mount('/content/drive')
   ```

Now when you open the *Files* window on the left, you'll be able to view `drive/MyDrive` which contains all your Google Drive files. You can run terminal commands by prefixing the command with an exclamation point in a cell. For example, if your code `helloworld.py` is in `mycode/`, then you can run:

```
!pwd
!cd /content/drive/MyDrive/mycode
!pwd
!python helloworld.py
```

Whether you're using Colab as a notebook or a terminal, if you want to use a GPU, you must set the Runtime to use a GPU via *Runtime* → *Change runtime type* → *T4 GPU*. Better GPUs are available by upgrading to Colab Pro.

Requirements.txt

For this assignment, we have provided you with a *requirements.txt* file which may be helpful to install dependencies for this homework. If you are running x86-64 Python on MacOS, this command should work out of the box:

```
pip install -r requirements.txt
```

For different machines/python versions, you may need to tweak the exact version of the dependencies to fit what is supported on your machine. We don't expect minor differences in versions to impact your ability to complete the assignment.

1 LaTeX Template Alignment (0 points)

1.1. (0 points) **Select one:** Did you use LaTeX for the entire written portion of this homework?

- ● Yes
- ○ No

1.2. (0 points) **Select one:** I have ensured that my final submission is aligned with the original template given to me in the handout file and that I haven't deleted or resized any items or made any other modifications which will result in a misaligned template. I understand that incorrectly responding yes to this question will result in a penalty equivalent to 2% of the points on this assignment. **Note:** Failing to answer this question will not exempt you from the 2% misalignment penalty.

- ● Yes

2 Background Reading (6 points)

This assignment is *primarily* a reading assignment. You will read both tutorials and the starter code.

Complete the following readings before you begin.

- PyTorch Tutorial. Please read the full collection of the Introduction to PyTorch, i.e. Learn the Basics || Quickstart || Tensors || Datasets & DataLoaders || Transforms || Build Model || Autograd || Optimization || Save & Load Model.

  https://pytorch.org/tutorials/beginner/basics/intro.html

- Weights & Biases Tutorial. Please read the Quickstart.

  https://docs.wandb.ai/quickstart

2.1. (2 points) How does PyTorch's autograd system compute gradients, and why is this feature essential for training neural networks?

PyTorch builds a *dynamic* computation graph during the forward pass, where tensors with `requires_grad=True` remember the operation that produced them through `grad_fn`. When I call `loss.backward()` it performs reverse mode automatic differentiation: the graph is traversed from the scalar loss back to the leaves, the chain rule is applied as vector Jacobian products, and the results are accumulated into each parameter's `.grad`. If $y = M(x;\theta)$ and $g_y = \frac{\partial J}{\partial y}$, autograd computes $\frac{\partial J}{\partial x} = \left(\frac{\partial y}{\partial x}\right)^\top g_y$ and $\frac{\partial J}{\partial \theta} = \left(\frac{\partial y}{\partial \theta}\right)^\top g_y$, summing contributions over all paths. By default the graph is freed after `backward()`, and I can keep it by setting `retain_graph=True`. This mechanism is essential because reverse mode AD delivers all parameter gradients for a scalar objective in time comparable to roughly one forward pass, removing the need for manual derivatives.

2.2. (2 points) Why is it beneficial to use something like wandb.log() over just printing the loss or accuracy every epoch?

Compared with printing, `wandb.log(...)` provides persistent, structured, time stamped records that do not vanish in console scrollback. It generates interactive plots automatically, enables side by side run comparisons, and supports logging media such as images, tables, confusion matrices, and model checkpoints that `print` cannot handle. I can search, filter, and group by metric, tag, or configuration, and step or epoch alignment keeps a coherent timeline even when I log asynchronously from different parts of the code. The cloud dashboard makes collaboration and sharing straightforward. Also it improves reproducibility by storing configuration and environment details, optionally the Git commit and random seed, and by allowing runs to resume as well as easy export to CSV or JSON.

2.3. (2 points) What do the MaxLen and PadSequence classes do in txt_classifier.py? Explain when and why each one is used.

MaxLen (TruncateToMaxLen). Cuts each tokenized sequence to a fixed maximum length `max_len`. *MaxLen* is applied per example before batching to bound sequence length.
PadSequence (PadToMaxLen). Right-pads sequences shorter than `max_len` with the special padding token (index `CorpusInfo.pad_idx`) so that all items in a batch share the same length. This is required to stack examples into a single tensor and to ensure downstream modules can ignore padded positions (via `padding_idx`, masks, or packing). *PadSequence* is applied at collate time so that batches have uniform shape and can be processed in parallel (batch size > 1).

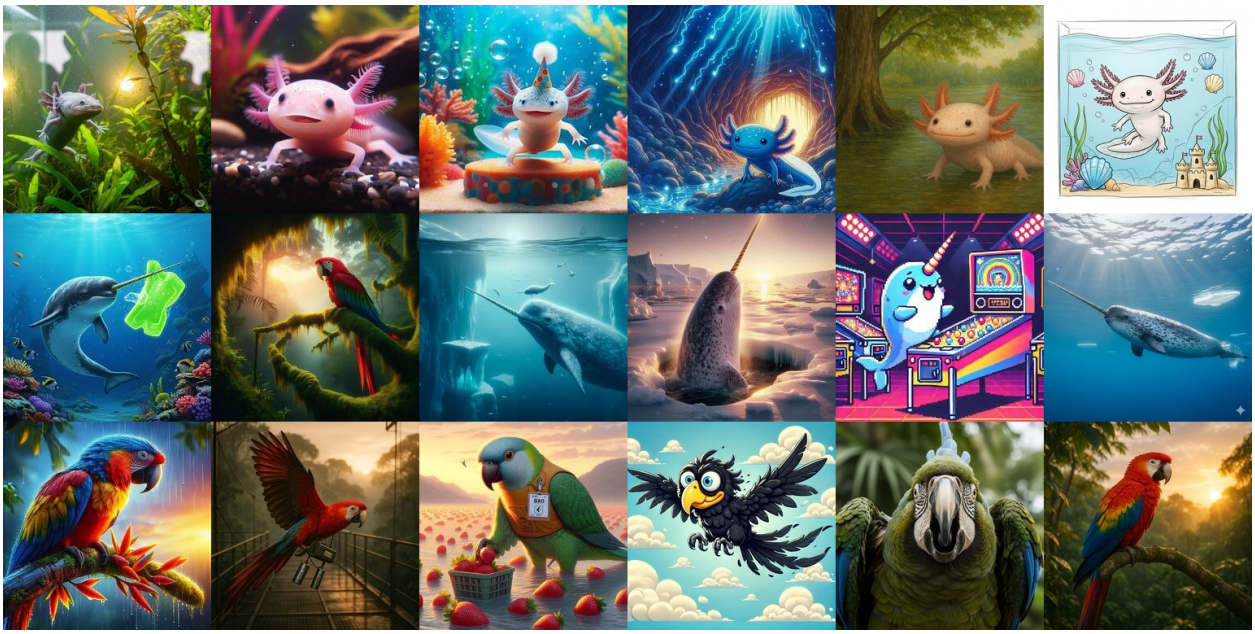


Figure 1: Examples images from Parrot/Narwhal/Axolotl dataset.

3 Image Classification (29 points)

In this section, you will augment an image classifier written in PyTorch.

Dataset

The image classification dataset is hot off the press: Each training example consists of an image created by a text-to-image model and is labeled as one of {parrot, narwhal, axolotl}. The dataset consists of:

`./data/img_train.csv` The training dataset. Each row corresponds to a training example. There are four columns: 1. The `image` column provides the relative path to the `.jpg` file (e.g. `./data/parrot/parrot-0.jpg`). 2. The `prompt` column contains the prompt that was fed into the text-to-image model to generate the image. 3. The `label` column is the human readable label (e.g. `narwhal`). 4. The `label_idx` column contains an integer representation of the label (i.e. 0, 1, or 2).

`./data/img_val.csv` The identically formatted validation dataset.

`./data/img_test.csv` The identically formatted test dataset.

`./data/parrot/ ./data/narwhal/ ./data/axolotl/` The three directories containing the respectively labeled raw images.

Examples of the images are shown in Figure 1. The data is split roughly so that the training data contains 70%, the validation data 15%, and the test data 15%.

Starter Code

The starter code in `img_classifier.py` is a fully functional (albeit simple) image classifier based on the PyTorch tutorial.

Data: The primary changes are to accommodate the Parrot/Narwhal/Axolotl dataset, intead of FashionMNIST. To accomplish this, we provide the class `CsvImageDataset` which is a `torch.utils.data.Dataset`. This class reads in the dataset from a given CSV, optionally applying some transformations to the image. The transformations we use in `get_data()` are fairly standard: first we resize so that the smallest side of the image is 256 pixels; next we crop out the centermost 256x256 pixels, then we normalize the red/green/blue channels using the mean and standard deviation from ImageNet.

Model: The model in `NeuralNetwork` is a simple feed forward neural network with ReLU activations. Its input layer takes the 256x256x3 numbers representing an image as features and maps to a first hidden layer of 512 units. The second hidden layer is also 512 units. The output layer is just 3 units, i.e. equal to the number of labels.

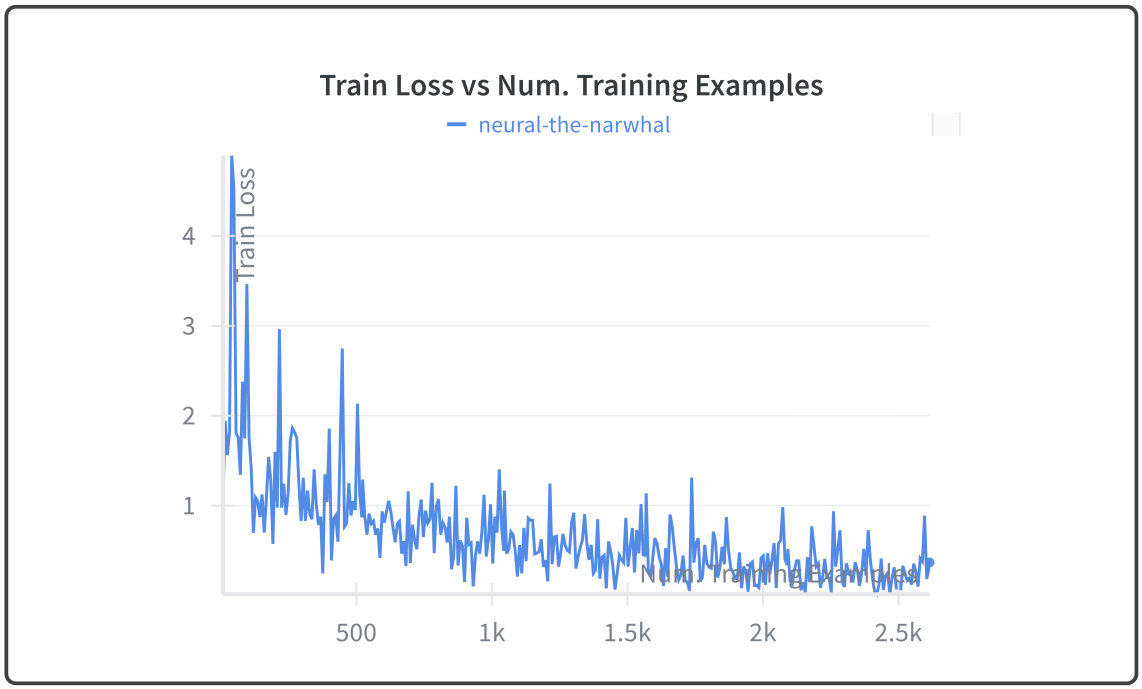
Training: Notice that our neural network does not explicitly define a `nn.Softmax` layer. The `nn.CrossEntropyLoss` works directly with the logits instead since the probabilities coming out of a softmax might be rather small. During training, we optimize the loss using the stochastic gradient descent algorithm, `optim.SGD`.

Evaluation: At the end of each epoch, the model is evaluated on the train. After training it is evaluated on the test dataset. Afterwards the model is saved to disk (and loaded by way of example) in case you wish to resume training from the saved state, or make predictions with the model on another dataset.

Experiments

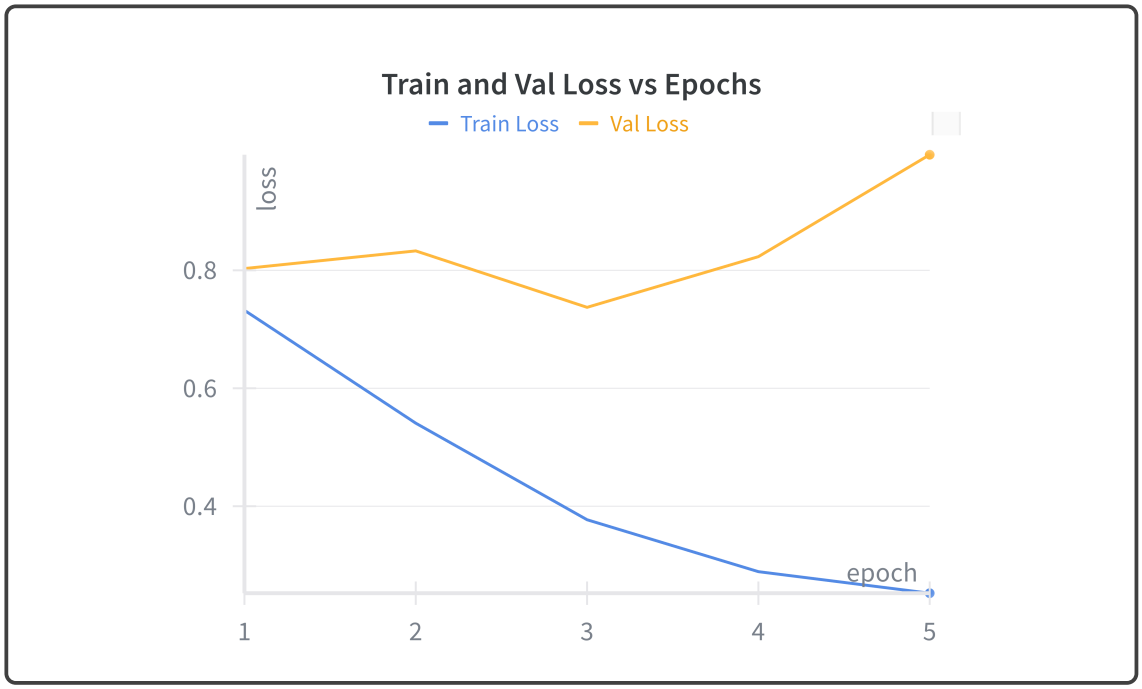
Instrument the existing code with `wandb`. First, use `wand.init()` to set the project name and store hyperparameters. Next use `wandb.log()` to track the loss of each batch and the current number of examples for which we've computed a gradient. Then, in a single call to `wandb.log()` at the end of each epoch, track the train accuracy, train loss, test accuracy, test loss, and the current epoch number.

3.1. (3 points) Run the image classifier with the default hyperparameters. Using `wandb`, plot the training loss of each batch vs. the total number of examples seen so far by the optimizer. The loss should be the average loss, taking the average within each batch. Name your run "neural-the-narwhal" and include the legend in your graph. (If you are having trouble figuring out how to enable a legend for a graph, please refer to this short tutorial video.)

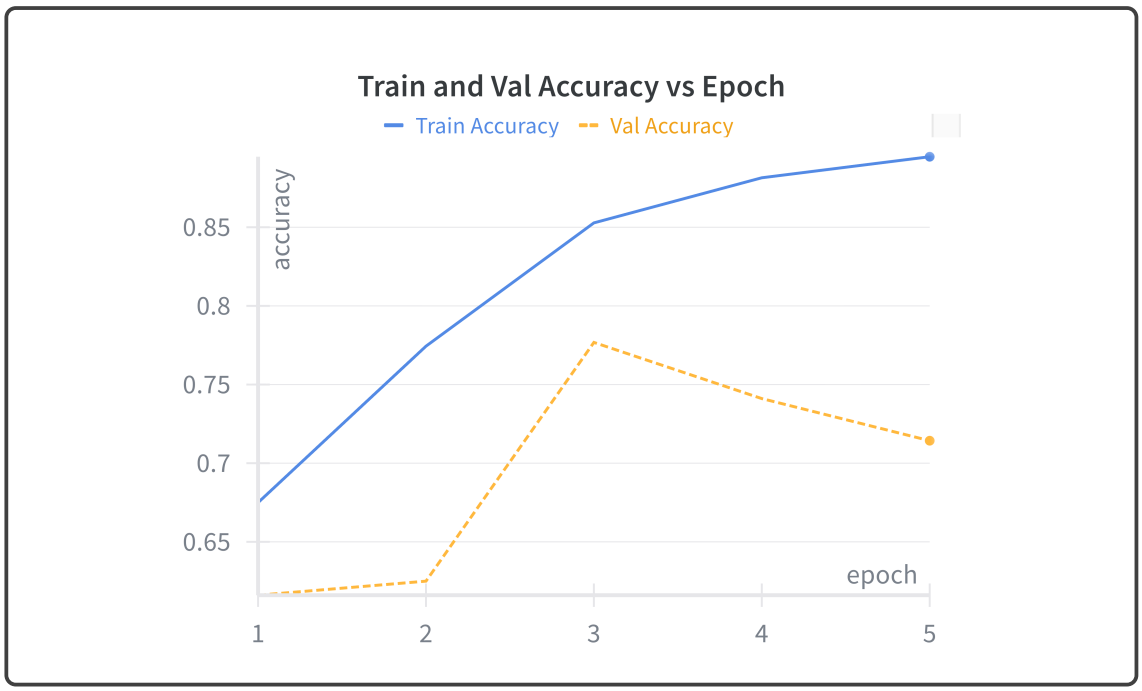


The dataset contains a validation set, but the starter code does not use it. Augment `img_classifier.py` so that the validation accuracy/loss are computed at the end of each epoch and reported to `wandb`.

3.2. (4 points) Using `wandb`, plot the train loss and the validation loss on the same plot with the number of epochs on the horizontal axis. The train loss should be averaged over the entire training dataset, and do likewise for the validation loss.



3.3. (4 points) Using `wandb`, plot the train accuracy and validation accuracy on the same plot with the number of epochs on the horizontal axis.



3.4. Surprisingly, this simple model does seem to be able to learn to distinguish (at least some of) the parrot/narwhal/axolotl images. Perhaps the reason for this success is that the colors of the animals tend to be quite different. Change the sequence of transforms in `transform_img = T.Compose([...])` so that it converts the image to a single channel grayscale image using `torchvision.transforms.Grayscale`. Be sure to adjust your model to accommodate the new input size! (Revert back to using color images after this question.)

3.4.a. (3 points) What is the test accuracy with color images vs. grayscale images?

Dataset	Image Type	Accuracy
Test	Color	69.027
Test	Grayscale	51.327

3.4.b. (2 points) Does this support the hypothesis that the classifier wasn't really learning anything besides the color of the animals? Provide justification for your answer.

No, the evidence suggests color helps but isn't the only cue. If the classifier relied *only* on color, removing chroma should drive accuracy to chance, i.e., $1/3 \approx 33.33\%$ for three classes. Our grayscale result is $51.327\% > 33.33\%$, which indicates the model is also exploiting color-invariant features e.g., silhouettes/edges, textures, and distinctive parts (parrot beak/head ratio, narwhal tusk, axolotl external gills). The drop from the color model (typically higher) quantifies color's contribution, but performance well above chance shows the network learns non-color structure as well.

3.5. For the first batch of each of the datasets (train, val, test), on the last epoch only, log each of the images with a caption containing its predicted label as a string (i.e. parrot, narwhal, axolotl) and its true label as a string. Format the caption as "`<predicted label> / <true label>`".

Because the image tensors have been normalized, you need to denormalize them before logging to wandb:

```
denormalize = T.Normalize(
    mean=[-m/s for m, s in zip(normalize_mean, normalize_std)],
    std=[1/s for s in normalize_std])
Xi_denorm = denormalize(X[i])
```

Then you can log them to wandb after converting them to the appropriate form:

```
wandb.Image(Xi_denorm, caption="", mode="RGB")
```

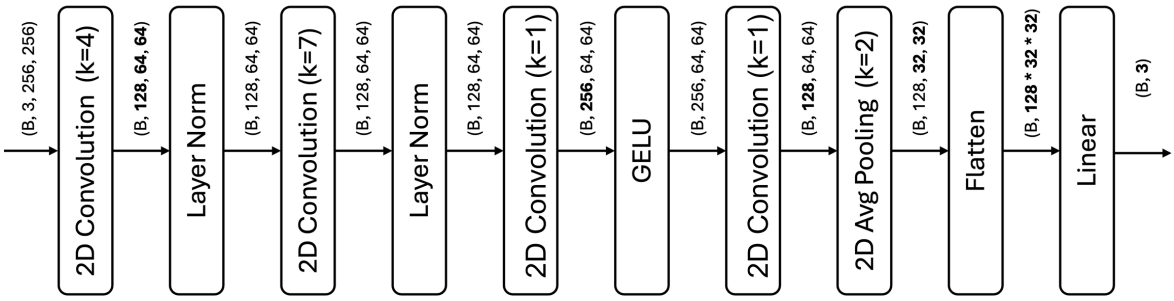
.

3.5.a. (2 points) Show the first batch of the test dataset captioned appropriately on wandb.[2]

3.5.b. (2 points) Pick a few of the images on which the model made an error, and try to explain why it might have had difficult with those images. Answer in just a few sentences.

Most errors were narwhal↔axolotl. Both appear in water with predominantly blue backgrounds, so background/texture priors are similar. In many mistakes aquatic context dominates making these two classes especially confusable.

3.6. The model we're using is incredibly simple: a feed-forward neural network with two hidden layers of size 512 units, and ReLU activations. Define a new model that implements the following computation graph:



The tuples denote the dimensions `(Batch Size, Channels, Img Height, Img Width)` of the output from the upstream layer and `k` denotes the kernel size.

You should review the documentation of `torch.nn`[3] and select corresponding modules for each layer of the graph.

Pay special attention to the dimensions in bold that change across layers to determine the correct parameters for each module. There may be more than one valid module or method of implementation for some layers.

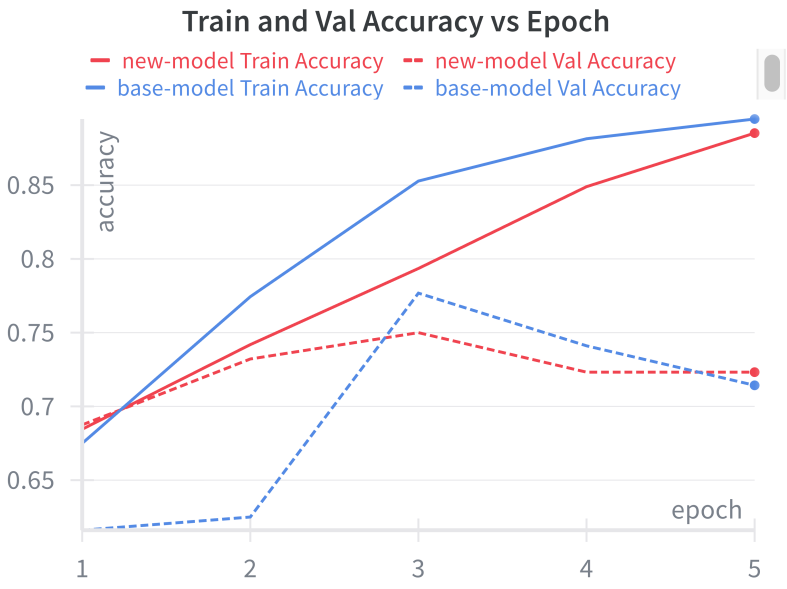
[2] Naturally, you *could* also do this a third time for the validation dataset, but we do not request that here.

[3] https://pytorch.org/docs/stable/nn.html

3.6.a. (1 point) Report the final train/test accuracy of your original model and your new model.

Dataset	Accuracy (Original Model)	Accuracy (New Model)
Train	89.484	88.528
Test	69.487	68.142

3.6.b. (4 points) With `wandb`, plot the train accuracy and validation accuracy **on the same plot** with the number of epochs on the horizontal axis. Include a run with the original model and a run with your new model. Name your runs "new-model" and "base-model" and include a legend.



3.6.c. (2 points) How many parameters were in your original model and new model? (Refer to the recitation handout for how to calculate these numbers.)

Original model

100928003

New model

3365507

3.6.d. (2 points) Did your new model perform better or worse on the test set than the original model? What about the training set? What factors might explain the differences in accuracy you observed between the models?

The CNN shows a stronger inductive bias for images i.e. via locality and weight sharing (translation equivariance) so with far fewer parameters it attains comparable test accuracy while keeping a smaller train–test (generalization) gap. By contrast, the dense baseline (MLP) uses $\sim 20\times$ more parameters, reaches higher training accuracy and can occasionally edge out test accuracy via sheer capacity, but it overfits more and exhibits a larger generalization gap (often latching onto background/color artifacts).

Title	Real Article	Fake Article
Vice Mayor of Wenzhou City Chen Yingxu and his delegation visited Ruian Middle School to investigate mental health education work	On November 27, 2023, Wenzhou Deputy Mayor Chen Yingxu and his delegation visited Ruian Middle School for investigation and guidance, and learned about the school's campus safety, mental health education and other work...	Wenzhou City, China - [Date] In a proactive move towards enhancing the mental well-being of students, Vice Mayor Chen Yingxu led a delegation to Ruian Middle School to delve into the intricacies of mental health education...
Bensalem Letter Carrier Honored For 43-Year Career	BENSALEM TOWNSHIP, PA —He was her mailman for 30 years. Every day, Debbie McBreen would see the man "who always had a smile on his face" —U.S. Postal Service letter carrier Kyle Livesay. "He was amazing, so friendly. He is just a wonderful person and an excellent man," said McBreen...	In a heartwarming ceremony held at the Bensalem Post Office, the community came together to celebrate the remarkable career of Mr. John Anderson, a dedicated letter carrier who has faithfully served the residents of Bensalem for an impressive 43 years...
No straight drive today as traffic to be diverted for T20I	Indore: As the city gears up for the much-anticipated T20I cricket match between India and Afghanistan on Sunday, Indore police released a traffic diversion plan to ensure a smooth commute during the match. DCP traffic Manish Agarwal said special arrangements have been made to ensure zero traffic block incidents on Sunday...	In anticipation of the throngs of cricket enthusiasts expected to flood the city for today's exciting Twenty20 International (T20I) match, city officials have announced a comprehensive traffic diversion plan. The match, set to be a showdown between two of the world's leading cricket teams...

Table 1: Examples of article snippets from the dataset.

4 Text Classification (19 points)

In this section, you will augment a text classifier written in PyTorch.

Dataset

> Note that the news articles, real and fake, in this dataset have not been filtered. Please take care when reading any of the data.

The text classification dataset consists of news articles, real and fake. The real articles were selected from news outlets around the world, with an emphasis on small-town local news. Each fake article was generated by a large language model with a prompt that included the title of the corresponding real news article.

The dataset is contained in three `.csv` files in UTF-8 encoding.

```
./data/txt_train.csv  ./data/txt_val.csv  ./data/txt_test.csv
```

Each one is identically formatted with one example per row. There are four columns:

1. The `article` column contains the text of the news article. The title of the article is prepended as the first line of this article text.
2. The `source` column differs for real/fake examples. For a real example, it contains the URL of a real article. For a fake example, it contains the prompt that was fed into the LLM to generate the article.
3. The `label` column is the human readable label (i.e. `real` or `fake`).
4. The `label_idx` column contains an integer representation of the label (i.e. 0 or 1).

Examples of the news articles are shown in Table 1. The data is split roughly so that the training data contains 70%, the validation data 15%, and the test data 15%. Each of the train/val/test CSV files are sorted so that the pairs of real/fake news articles are adjacent rows.

Starter Code

The starter code in `txt_classifier.py` is a fully functional (albeit simple) text classifier.[4]

Data: The dataset for this section is starts from raw text and makes several important transformers to convert it into a tensor representation. The starter code provides the class `CsvTextDataset` which is a `torch.utils.data.Dataset`. This class reads in the dataset from a given CSV, optionally applying some transformations to the text. We read the training dataset twice: the first time we tokenize the text but do *not* numericalize the tokens in order to build a vocabulary (i.e. a mapping of tokens to integers). The second reading of the training data then proceeds with numericalization. Each article is also truncated to some maximum number of tokens. The `DataLoader` uses a `collate_fn`, which is simply a callable (or function) that is applied to each batch. You will explore the behavior of our `PadSequence` in the questions below.

Model: The `TextClassificationModel` consists of just two layers. The first layer one uses `nn.EmbeddingBag` which applies an `nn.Embedding` layer to convert each token into a word embedding and then performs average pooling to combine an

[4] Our source code is based on an old PyTorch tutorial for text classification, which you are welcome to review.

arbitrary number of embedding vectors into a single one. The second layer is just a `nn.Linear` layer to produce an output layer where the number of output units equals the number of classes.

Training: The training process proceeds as usual: optimizing a cross entropy objective with SGD. Note that `nn.CrossEntropyLoss` assumes its inputs are *unormalized logits* – that is, the last layer of our model does *not* include a softmax to produce a probability distribution. Instead, the cross-entropy is computed more stably / efficiently by working directly on the inputs to the (hypothetical) softmax.

Evaluation: The model periodically evaluates on the train and validation set, and finally on the test set.

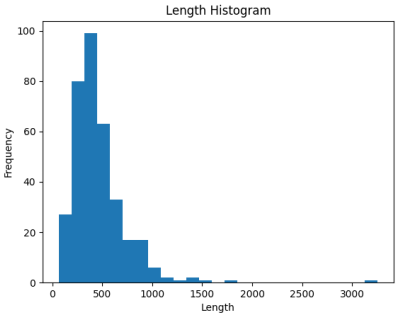
Experiments

(You are welcome, but not required, to instrument the code with `wandb`. Assuming you do, be sure to select a new project name (e.g. `wandb.init(project='txt_classifier', ...)`).

4.1. (3 points) Generate a histogram of the lengths of the raw news articles *without* truncation or padding. You can do this however you'd like. (We recommend using `matplotlib`. If doing so, follow the documentation from here.)

To run without truncation / padding, you should use:

```
python txt_classifier.py --batch_size 1 --max_len -1 --length_histogram
```



4.2. Carefully read the documentation for `nn.Embedding`[5] and then `nn.EmbeddingBag`[6] paying close attention to the `padding_idx` parameter and the associated examples.

Next implement a truncation transform `TruncateToMaxLen` that cuts each article to `args.max_len`. Then implement a padding transform `PadToMaxLen` that uses the `CorpusInfo.pad_idx` field to pad each article to `args.max_len`.

4.2.a. (1 point) Now perform the runs below of the text classifier and compare the runtime of each run. Using `--max-len -1` will disable the truncation / padding transforms you wrote.

```
python txt_classifier.py --batch_size 1 --max_len -1
python txt_classifier.py --batch_size 1 --max_len 1024
python txt_classifier.py --batch_size 32 --max_len 1024
```

Batch Size	Max Length	Runtime (seconds)
1	None	2.84
1	1024	2.78
32	1024	1.38s

4.2.b. (1 point) Now do one more run and report the error that you get.

```
python txt_classifier.py --batch_size 32 --max_len -1
```

```
RuntimeError: stack expects each tensor to be equal size (got
[258] at entry 0 and [141] at entry 1)
```

4.2.c. (2 points) Why does this last run yield this error?

I'm getting `RuntimeError: stack expects each tensor to be equal size (got [258] at entry 0 and [141] at entry 1)` because the `DataLoader` is trying to `torch.stack` token tensors of unequal lengths into a batch; with `--max_len -1` I disabled truncation/padding, so sequences keep their raw lengths.

[5] https://pytorch.org/docs/stable/generated/torch.nn.Embedding.html
[6] https://pytorch.org/docs/stable/generated/torch.nn.EmbeddingBag.html

4.3. (2 points) Run the code with the existing SGD optimizer and with the given learning rate (5). Then run again but switch the optimizer to `torch.optim.Adam`[7] using Adam's default learning rate (0.001). For both runs, report the validation and test accuracies. (You can easily create such a table on the "Runs" tab of `wandb` if these are logged and you log the optimizer name.) (Do NOT revert this change, but use Adam for future questions as well.)

	Val. Acc.	Test Acc.
SGD	50.00	50.00
Adam	59.21	60.81

4.4. Replace the simple `TextClassificationModel` with an LSTM-based classifier. This can be any architecture of your choosing, so long as it uses `torch.nn.LSTM`[8] in some reasonable way (e.g. unidirectional LSTM, bidirectional LSTM, deep LSTM). Please read through the PyTorch documentations for sequential models and LSTM for more examples and clarifications on implementation details.

In your implementation, you must use `nn.Embedding` instead of `nn.EmbeddingBag`. The reason for this is that `nn.EmbeddingBag` performs operations such as sum, mean, or max on the input sequences, which destroys the sequential nature of the data that LSTMs are designed to process. This loss of sequence information effectively reduces it to a multi-layer perceptron (MLP).

Our data preprocessing puts the batch dimension first. However, for historical reasons `nn.LSTM` assumes by default the sequential dimension (i.e. the dimension corresponding to the token positions) is first. You should use the `batch_first=True` option on `nn.LSTM` to ensure your code behaves correctly; see the LSTM documentation for additional details.

Your model should incorporate a pooling layer after the LSTM outputs. While using only the last timestep output may seem sufficient, it can be adversely affected by padding tokens, which are often present in the sequences. To mitigate this issue, it is advisable to add a pooling layer to the LSTM outputs (e.g., `nn.AdaptiveMaxPool1d`) before passing them to the linear layers.

Additionally, retain the Adam optimizer in your implementation. Ensure that you consistently use the custom padding tokens defined in `CorpusInfo`. Maintaining this consistency is essential for effective embedding and sequence processing.

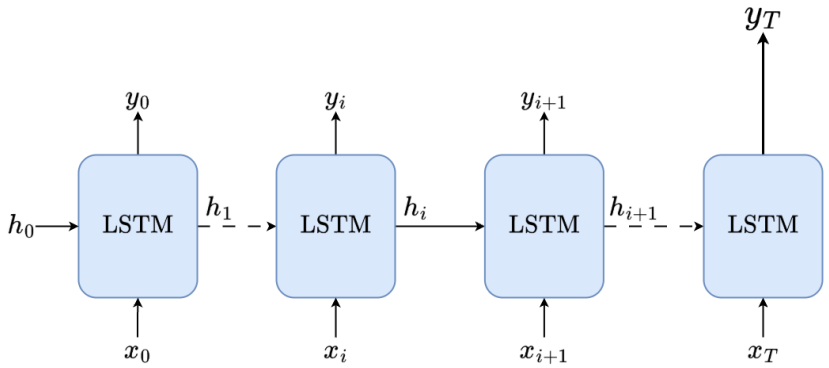
Lastly, while your goal is to practice using LSTMs for text processing and understand their capabilities with sequential inputs, there is no requirement for your model to outperform the baseline model.[9]

[7] https://pytorch.org/docs/stable/generated/torch.optim.Adam.html

[8] https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html

[9] Historically, we've found that whether our reference LSTM implementation beats the baseline classifier seems to depend on the data created in that semester.

4.4.a. (2 points) In math, pseudocode, or a computation graph, describe the structure of your new model.



4.4.b. (4 points) Report the validation and test accuracies of your new model against the original model. The horizontal axis should be the number of epochs. Your results do not have to be better than the original model.

	Val. Acc.	Test Acc.
Original Model	59.211	60.811
LSTM Model	72.368	78.378

4.4.c. (2 points) Describe any differences in the train and validation accuracies between your model and the original model. In a few sentences, discuss why your model worked better or worse than the original model. Consider how input sequences are processed in an LSTM and how the presence of padding tokens might affect its learning.

Compared to the original `TextClassificationModel`, the LSTM attains higher accuracy on train (99.4% vs. 67.4%) and validation (72.3% vs. 59.2%) but exhibits a much larger train–validation gap (27.1 vs. 8.2 points), signaling stronger overfitting. The improvement stems from sequence modeling: the LSTM preserves word order and captures dependencies (e.g., negation), whereas `EmbeddingBag` averages embeddings and discards order. Padding remains a caveat: although `padding_idx` maps pad tokens to zero vectors, the LSTM still processes these time steps, which can dilute the hidden state for short sequences due to repeated application of non-linearity and W_{hh} on the hidden state.

4.5. (2 points) Look through a few of the real/fake news article pairs. In a few sentences, speculate on how the model is able to train successfully.

I believe the model separates real from fake news by exploiting a cluster of surface regularities that correlate with the label. Fake pieces often adopt a conversational, assistant like tone, with closing phrases such as "Would you like me to…" or "Have a tip?", whereas real pieces maintain a neutral journalistic voice. Formatting also differs in practice, since synthetic articles frequently include bracketed link patterns, emoji headers, lists, and stylized section banners, while real articles keep conventional newsroom layout. The overall structure diverges as well, because fabricated stories often add meta commentary or summaries such as "Here is a summary," "In summary," or "Bottom line," whereas real reports present facts directly without this framing. Source attribution supplies another cue, as genuine articles reference outlets and reporters inline, for example "told WTOP" or "NEWS10 reported," while fabricated texts lean on formal URL style citations and numbered references. Length and density contribute too, with synthetic items tending to be verbose and repetitive, while real items are concise and focused on essential facts.

5 Code Upload (0 points)

5.1. (0 points) Did you upload your code to the appropriate programming slot on Gradescope?
Hint: The correct answer is 'yes'.

- ● Yes
- ○ No

For HW0, you should upload *two* files: `img_classifier.py` and `txt_classifier.py` that include all your interesting new changes to the code.

6 Collaboration Questions (2 points)

After you have completed all other components of this assignment, report your answers to these questions regarding the collaboration policy. Details of the policy can be found in the syllabus.

6.1. (1 point) Did you collaborate with anyone on this assignment? If so, list their name or Andrew ID and which problems you worked together on.

No

6.2. (1 point) Did you find or come across code that implements any part of this assignment? If so, include full details.

No