Instructor: Alina Vereshchaka

Assignment 1

CNN, RNN and LSTM Architectures

Checkpoint: Feb 25, Tue, 11:59 pm

Due date: Mar 6, Thu 11:59pm

Welcome to Assignment 1! This assignment focuses on developing both theoretical and practical skills related to convolutional neural networks (CNNs), recurrent neural networks (RNNs) and Long Short-Term Memory (LSTM) networks. It consists of four hands-on exercises where you will analyze network setups, work with various datasets, and implement, train, and adjust different neural networks. There is also a theoretical component covering CNNs and LSTMs.

- All libraries are allowed, except those with pre-trained or predefined models. Submissions with pre-trained predefined architecture will not be considered for evaluations, except for the Bonus part related to Transfer learning.
- The assignment is expected to be fully completed using PyTorch (recommended) or Tensorflow. Mixing is not allowed.

Part I: From VGG to ResNet [30 pts]

Implement and compare one of the fundamentals CNN architectures: VGG-16 (Version C) and ResNet-18 for image classification. Explore advanced techniques to improve model performance and explore the transition from standard deep CNNs to networks with residual connections.

The expected accuracy is above 75% for base model and 80% for improved model.

CNN Dataset



A dataset containing 30,000 images (10,000 images per class) of dogs, cars, and food. Each image is 64x64 pixels. Located at UBlearns > Assignment 1

Step 1: Data preparation

1. Load CNN dataset (UBlearns > Assignment 1).
    - Analyze the dataset, e.g., return the main statistics. Provide a brief description (2-3 sentences) of the dataset: What does it represent / key features.
2. Create at least three different visualizations to explore the dataset. Provide a short description explaining what each visualization shows. Examples include:
    - Display a grid of sample images from each class (e.g., a 3x3 grid showing 3 dogs, 3 cars, and 3 food items
    - Bar chart showing the number of images per class
    - Histogram of pixel values for a single channel of a few sample images. This helps understand the data's distribution and choose appropriate normalization
    - Calculate and display the "average" image for each class. This can reveal common features.
3. Preprocess the dataset.
    - Normalize the pixel values to the range [0, 1]. E.g. divide by 255.0
    - [If needed] Address class imbalance in the target column. Possible solutions: oversampling; undersampling; data augmentation techniques for the minority class; assign higher weights to the minority class and lower weights to the majority class, etc.
    - Convert target variable needs to numerical format. You can use one-hot encoding. However, if you use torch.nn.CrossEntropyLoss for your network, it expects class indices (0, 1, 2) directly, not one-hot encoded vectors. Therefore, ensure your labels are integer tensors (e.g., torch.LongTensor).
4. Split the dataset into training, testing and validation sets. You can use train_test_split from scikit-learn.

Step 2: Implementing VGG

1. Implement the VGG-16 (Version C) architecture following the proposed architecture and adjusting it to fit our dataset. Refer to the original VGG paper for more details. You need to make few adjustments to the original VGG strcucture based on your dataset:
    a. The input size for the original VGG is 224x224. Here are a few options:
        - You can remove a few max pooling layers (e.g the last two max-pooling layers)
        - Add padding to your convolutional layers to maintain more spatial dimensions.
        - Resize your input size
    b. The last layer of the original VGG contains 1000 classes. You can modify it to fit the number of your classes (e.g. 3).

2. Use the dropout and learning rate scheduler, as in the VGG paper (Chapter 3.1 Training). Experiment with at least two different weight initialization strategies (e.g., Xavier/Glorot, He initialization). Report the impact on training.
3. Train your VGG-16 model on a dataset:
    a. Experiment with at least three different optimizers (e.g., SGD, Adam, AdamW, RMSprop). Compare their performance and discuss your findings.
    b. Experiment with at least two different batch sizes (e.g., 32, 64, 128). Report the impact on training speed (time per epoch) and performance.

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

4. Regularization and overfitting prevention. Apply at least three techniques to prevent overfitting and improve the results. Discuss each of the techniques you have used and how they impact performance. You can consider the following methods: regularizations, early stopping, image augmentation.
5. Evaluation and analysis. Include the following metrics:
    a. Training accuracy, training loss, validation accuracy, validation loss, testing accuracy, and testing loss. Provide your short analysis.
    b. Plot the training and validation accuracy over time (epochs).
    c. Plot the training and validation loss over time (epochs).
    d. Generate a confusion matrix for the test set using sklearn.metrics.confusion_matrix. Display it using seaborn.heatmap. Analyze the confusion matrix: Which classes are most often confused?
    e. Calculate and report other evaluation metrics such as precision, recall and F1 score (details). Refer to sklearn.metrics.precision_recall_fscore_support. Analyze the results.
    f. Use TensorBoard (or a similar tool, e.g. Wandb) to log the training and validation loss and accuracy over epochs, generate the charts, and attach the SVG images of the charts.
    g. Display a few misclassified images from the test set along with their predicted and true labels. Try to explain why the model might have made these mistakes.

6. Save the weights of the trained network that provides the best results. Saving and loading models (PyTorch)

Step 3: Implementing ResNet

1. Implement residual blocks of ResNet, including convolutional layers, batch normalization, ReLU activation, and residual connections. You can use nn.Conv2d, nn.BatchNorm2d, nn.Sequential, nn.Identity.
2. Assemble the ResNet-18 architecture using your residual block implementation. Follow a configuration with 18-layer original ResNet paper.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

3. Train the ResNet-18 model on the same dataset, using the same data splits, preprocessing, and similar training procedures as in Step 2.
4. Apply the same regularization techniques that you found to be most effective for VGG-16.
5. Evaluation and analysis.
   a. Perform the same comprehensive evaluation as in Step 2, Part 5 (metrics, plots, confusion matrix, precision/recall/F1, TensorBoard, misclassified examples). Provide your analysis.
   b. Create at least two plots that directly compare the performance of ResNet-18 and VGG-16. Use different colors and legends to distinguish the models. Provide your analysis. For example, a plot showing:
      - Training accuracy of both models vs. epoch.
      - Validation accuracy of both models vs. epoch.
      - Training loss of both models vs. epoch.
      - Validation loss of both models vs. epoch.
6. Save the weights of the trained network that provides the best results. Saving and loading models (PyTorch)

Step 4: Discussion and conclusion

1. Briefly explain the theoretical concepts behind VGG and ResNet:
    - VGG: Discuss the idea of using small convolutional filters (3x3) and stacking multiple layers to increase depth.
    - ResNet: Explain the problem of vanishing/exploding gradients in very deep networks and how residual connections address this issue. Explain the concept of "identity mapping."
2. Discuss how the regularization and optimization techniques you used impacted the performance of both models. Refer to your experimental results (plots, metrics).
3. Analyze the results, including all the required graphs and metrics (as listed in Step 2, Part 5, and Step 3, Part 5). Focus on the comparison between VGG-16 and ResNet-18. Which model performed better? Why?
4. Summarize your findings, highlighting the key differences between VGG-16 and ResNet-18. Discuss the advantages of residual connections in deep CNNs, based on your experimental results. State which model you would recommend for this task and why.
5. **References.** Include details on all the resources used to complete this part.

Notes:

- Works with in-build vgg, resnet or pretrained models won't be considered for evaluation.
- You can partially reuse related implementations that you have completed for other courses with a proper citation, e.g. "Part I, Step 1 is based on the CSE 676 Deep Learning Assignment 0 submission by My Full Name"
- For this part, there is no need to include the report. Submit a single, well-organized Jupyter Notebook containing all code, results, visualizations, and discussion. Make sure the notebook runs without errors. Use markdown cells to clearly separate sections and provide explanations.
- Use torch.manual_seed() and random.seed() for reproducibility of your results.

Part II: Investigating the Vanishing Gradient Problem [20 pts]

Experimentally demonstrate the vanishing gradient problem in deep CNNs and understand how ResNet's architecture mitigates it. You will also explore other key CNN concepts through additional experiments.

STEPS:

1. Create a deeper version of your VGG-16 network defined in Part I, named "VGG-Deep".
   - Add four additional convolutional layers to the feature extraction part of your VGG-16 network (i.e., before the fully connected layers). Maintain the pattern of increasing the number of filters as you go deeper. Do not add any pooling layers beyond what you already have in your adapted VGG-16
   - For this experiment, remove any batch normalization, dropout, or other regularization techniques from VGG-Deep. We want to observe the effect of depth in isolation.
   - Keep all other aspects of the network (kernel sizes, strides, padding, activation functions – which should be ReLU) identical to your original VGG-16.
2. Training VGG-Deep:
   - Train the "VGG-Deep" network on the same dataset and with the same preprocessing and data augmentation as your original VGG-16.
   - Use a simple optimizer like SGD (without momentum). This simplifies the analysis and highlights the vanishing gradient issue more clearly.
   - Carefully observe the training behavior (loss curves, accuracy). You should expect to see slower learning and potentially lower accuracy compared to your original VGG-16.
3. Gradient analysis:
   a. During training, track the average L2 norm of the gradients in each convolutional layer. Implement this using PyTorch hooks:
      - Use register_full_backward_hook on each convolutional layer's weight parameter (not the bias). This is the correct hook to use for gradient analysis. Example:

```
def gradient_hook(module, grad_input, grad_output):
      # grad_output is a tuple, and we want the gradient w.r.t.
      weights
      grad_norm = grad_output[0].norm(p=2).item()
      # Store grad_norm (e.g., in a list or dictionary)
      # You'll need a way to associate this norm with the layer and
      iteration

for layer in vgg_deep.features: # Assuming 'features' holds the conv
layers
      if isinstance(layer, nn.Conv2d):
            layer.weight.register_full_backward_hook(gradient_hook)
```

      - Inside the hook, calculate the L2 norm (Euclidean norm) of the gradient using *.norm(p=2)*. Convert it to a Python scalar using *.item().*
      - Store these gradient norms in a data structure (e.g., a list of lists, a dictionary). You need to keep track of the norm for each layer and each training iteration (or like every 10th iteration, to reduce memory usage).

b. Plots:
   - Create a plot showing the average gradient norm for each convolutional layer over time (training iterations or epochs). Each layer should have its own line on the plot. Use a different color for each line and include a clear legend.
   - Create a separate plot that shows the gradient norms for a subset of layers (e.g., the 2nd, 5th, 8th, and the deepest convolutional layer). This is for the direct comparison of how gradients change across different depths. Place all these lines on the same plot.

c. You should observe that the gradient norms decrease significantly as you move deeper into the network (earlier layers have smaller gradient norms), demonstrating the vanishing gradient problem.

4. Comparison with VGG-16 and ResNet-18:

   a. Compare the training curves (loss and accuracy vs. epoch) of VGG-Deep, your original VGG-16 (from Part I), and your ResNet-18 (from Part I). Create a single plot for loss and a single plot for accuracy, showing all three models. Use clear legends and different colors.

   b. While you don't need to recompute the gradients on Resnet, discuss how ResNet's residual connections are expected to impact the gradient flow compared to VGG-Deep.

5. Investigate and analyze more setup. Use your VGG implementation as a base for these, but create new, smaller models for experimentation to keep training times manageable. Select any THREE:

   a. Impact of kernel size. Create small VGG-like networks (e.g., 3-4 convolutional layers) with different kernel sizes (3x3, 5x5, 7x7). Keep the number of filters, strides, and padding consistent. Train them on a subset of the data (e.g., 1000 images per class) to speed up experimentation. Compare their performance (accuracy) and discuss how kernel size affects the receptive field and the ability to capture features.

   b. 1x1 Convolutions. Explain the purpose and benefits of 1x1 convolutions (dimensionality reduction, feature recombination). Implement a small network (e.g., 2-3 convolutional layers) with and without 1x1 convolutions. Compare their performance (accuracy), parameter count (using torchsummary.summary or a similar method), and discuss the trade-offs.

   c. Max Pooling vs. Average Pooling. Create a small VGG-like network. Train two versions: one with max pooling and one with average pooling. Compare their performanmce (accuracy) and discuss when one might be preferred over the other. Consider situations where preserving spatial information is important vs. situations where robustness to small variations is desired.

   d. Activation functions. Experiment with different activation functions (ReLU, Leaky ReLU, ELU, GELU) in a small VGG-like network. Compare their performance (accuracy, convergence speed). Discuss the properties of each activation function (e.g., how they address the dying ReLU problem).

e. Learned filter visualization. Visualize the learned filters (weights) of the first convolutional layer of your trained VGG-16 and ResNet-18 models from Part I. Use matplotlib.pyplot.imshow to display the weights. Are there any discernible patterns? Do the filters look different between VGG-16 and ResNet-18?

f. Computational cost and memory. Compare the theoretical computational cost using Big O notation and the memory required for VGG-16, VGG-Deep, and ResNet-18. Use torchsummary to confirm practically the number of parameters.

6. Analysis and discussion:

a. Analyze your gradient norm plots. Do they demonstrate the vanishing gradient problem? Explain how the gradient norm changes as you move deeper into VGG-Deep. Be specific and quantitative (e.g., "The gradient norm of layer 2 is X times larger than the gradient norm of layer 10").
b. Explain why the vanishing gradient problem occurs in deep networks. Relate this to the backpropagation algorithm and the chain rule. Discuss how the repeated multiplication of small gradients can lead to extremely small values in earlier layers.
c. Explain how ResNet's architecture (residual connections) helps alleviate the vanishing gradient problem. Explain how the identity mapping allows gradients to flow more easily through the network.
d. Discuss the theoretical impact of batch normalization on the vanishing/exploding gradient problem. Explain how it helps stabilize and accelerate training.
e. Summarize the key findings from your three chosen investigations.
f. **References.** Include details on all the resources used to complete this part.

Notes:

- For this part, there is no need to include the report. Submit a single, well-organized Jupyter Notebook containing all code, results, visualizations, and discussion. Make sure the notebook runs without errors. Use markdown cells to clearly separate sections and provide explanations.
- Use torch.manual_seed() and random.seed() for reproducibility of your results.

Part III: Time-Series Forecasting using RNNs [20 pts]

In this part, we work on time-series forecasting using RNN and LSTM methods. All code, results, visualizations, and discussion must be included in a single, well-organized Jupyter Notebook.

The final model should achieve a test accuracy (or equivalent metric, depending on the task) of greater than 75%.

DATASETS

Choose ONE dataset from the following real-world time-series datasets:

- Energy Consumption: electricity consumption data collected from multiple households
- Air Quality Dataset: hourly air quality data with various features like temperature, humidity, and gas concentrations
- Web Traffic Dataset: number of visits or page views to websites
- Electric Grid Stability Dataset: stability status of an electrical grid
- Taxi Trip Duration Dataset: taxi trip durations with features like pickup/dropoff coordinates, timestamps, and other related information.
- Retail Sales Dataset: sales figures of a retail store or chain over time

Step 1: Data exploration and preprocessing.

1. Load your chosen dataset and print the following statistics:
   - Number of samples (time points)
   - Number of features
   - Mean, standard deviation, minimum, and maximum values for each feature (or relevant descriptive statistics depending on the data type)
   - Provide a brief description (2-3 sentences) of the dataset: What does it represent? Where does it come from (provide a link)? What are the key variables?
2. Identify any missing values (e.g. using pandas.isnull().sum()).
3. Handle any missing values (imputation or removal). Common imputation methods include: forward/backward fill, mean/median imputation, linear interpolation).
4. Create at least three different visualizations to explore the dataset. Provide a short description (1-2 sentences) for each. Examples include:
   - Time series plot of the target variable (the variable you want to forecast).
   - Time series plots of individual features
   - Histograms or box plots of feature distributions
   - Correlation matrix heatmap (using seaborn.heatmap) to show relationships between features
   - Autocorrelation plot (using pandas.plotting.autocorrelation_plot) to identify seasonality or trends.
5. Normalize or standardize your data using appropriate techniques (e.g., MinMaxScaler, StandardScaler). Fit the scaler only on the training data, and use the fitted scaler to transform the validation and test sets.
6. If necessary, convert categorical features to numerical representations (e.g., one-hot encoding or label encoding).
7. Split the data into training, validation, and testing sets. For time-series data, the split must be sequential. Do not use train_test_split with shuffling. Instead, split the

data based on indices. For example:

```
train_size = int(len(data) * 0.7)
val_size = int(len(data) * 0.15)
train_data = data[:train_size]
val_data = data[train_size:train_size + val_size]
test_data = data[train_size + val_size:]
```

8. Sequence Creation:
   a. Create input sequences and corresponding target values for your RNN/LSTM. This involves creating "windows" of past data to predict the next time step.
   b. Choose a sequence length (e.g., 10, 20, 50). This is a hyperparameter. Explain your choice based on the characteristics of your data (e.g., seasonality, expected dependencies). Start with a reasonable value and experiment.
   c. Create a function to generate sequences. Example:

```
def create_sequences(data, seq_length):
    xs = []
    ys = []
    for i in range(len(data) - seq_length):
        x = data[i:(i + seq_length)]
        y = data[i + seq_length]  # Predicting the next value
        xs.append(x)
        ys.append(y)
    return np.array(xs), np.array(ys)
```

   - Apply this function to your training, validation, and test sets.
   - Ensure your sequences are NumPy arrays or PyTorch tensors.

Step 2: Model development

1. Choose either an RNN or LSTM network for your time-series forecasting task. Model architecture:
   - It should consist of at least three RNN/LSTM layers.
   - Experiment with different architectures: stacked RNNs/LSTMs, bidirectional RNNs/LSTMs, or a combination. You can use RNN/LSTM layers, e.g. torch.nn.RNN
   - Include appropriate activation functions (e.g., ReLU, tanh). Tanh is common for the recurrent layers, and ReLU can be used for intermediate layers.
   - Use dropout (torch.nn.Dropout) for regularization. Justify your chosen dropout rate (e.g., 0.2, 0.5). Experiment with different values.
   - Consider adding one or more fully connected layers (torch.nn.Linear) after the RNN/LSTM layers to produce the final prediction. This is often necessary to map the RNN/LSTM output to the desired output dimension.
   - Print model summary using torchinfo.summary
2. Train your model using an appropriate loss function (e.g., MSE) and optimizer (e.g., Adam, SGD).
   a. Tune hyperparameters (e.g., learning rate, batch size, number of epochs,

hidden layer size (for RNN/LSTM), dropout rate, sequence length) using your validation set.
   b. Describe your hyperparameter tuning strategy (e.g., grid search, random search, manual tuning). Document the a few hyperparameter combinations you tried and their results (e.g., in a table or using comments).
   c. Implement early stopping to prevent overfitting.
3. Save the weights of the trained neural network that provides the best results. Check saving and loading models (PyTorch)

Step 3: Evaluation and analysis

1. Evaluate your trained model on the test set. Report the following metrics:
   - Training accuracy/loss
   - Validation accuracy/loss
   - Testing accuracy/loss
2. Depending on your chosen dataset and task (regression or classification), report relevant metrics like:
   - Regression: Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), R-squared (coefficient of determination). Use sklearn.metrics.
   - Classification: Precision, Recall, F1-score, Confusion Matrix (using sklearn.metrics.confusion_matrix and seaborn.heatmap).
3. Provide the following plots:
   - Plot training and validation accuracy/loss curves over epochs.
   - Regression: plot predicted vs. actual values on the test set. A scatter plot or a line plot comparing the time series can be useful.
   - Classification: display the confusion matrix.
   - Use tensorboard to log training and validation metrics.
4. Discuss and analyze:
   - Briefly reiterate the key characteristics of your dataset.
   - Describe your final RNN/LSTM architecture in detail (number of layers, types of layers, hidden size, dropout rate, etc.).
   - Discuss your results, referencing the metrics and visualizations. Did your model achieve the expected accuracy? What were the challenges? How did hyperparameter tuning affect performance? Are there any patterns in the errors (e.g., consistent under- or over-prediction)?
   - Discuss any limitations of your model or approach.
   - Suggest potential improvements or future work (e.g., trying different architectures, incorporating more features, using a more sophisticated data preprocessing pipeline).
5. **References**. Include details on all the resources used to complete this part.

Part IV: Sentiment analysis using LSTM [20 points]

In this part, we perform a sentiment analysis using LSTM model. The final (improved) model should achieve a test accuracy of greater than 75%.

DATASETS

Choose ONE dataset from the following options for sentiment analysis:

- Amazon Product Reviews Dataset: reviews of various products sold on Amazon, labeled with sentiment.
- Yelp Reviews Dataset: reviews from the Yelp platform, labeled with sentiment
- Twitter Airline Sentiment Dataset: tweets about airline experiences, labeled with sentiment.
- Stanford Large Movie Review Dataset movie reviews classified as positive or negative
- Medical Dialogue Dataset: contains conversations between doctors and patients. It has 0.26 million dialogues
- News Headlines Dataset: news headlines labeled with sentiment, covering various news topics and categories.

Step 1: Data exploration and preprocessing.

1. Load your chosen dataset and print the main statistics.
2. Print the first 5 rows of the dataset to understand its structure.
3. Provide a brief description (2-3 sentences) of the dataset: What does it represent? What are the sentiment labels? Provide a link to the source.
4. Display descriptive statistics:
   - Number of samples.
   - Class distribution (percentage of positive, negative, etc. reviews).
   - Average review length (in words or characters).
   - Vocabulary size (number of unique words).
5. Handle missing values (if any). Explain your chosen method (e.g., removal, imputation – though removal is usually fine for text data if there are few missing values).
6. Create at least three visualizations to gain insights into the data:
   - Histogram of review lengths (both in words and characters).
   - Generate word clouds for *each* sentiment class. Use the wordcloud library (pip install wordcloud). This helps visualize the most frequent words associated with each sentiment.
   - Bar chart showing the number of reviews for each sentiment class.
   - Any other relevant visualization (e.g., n-gram frequency distributions).
7. Data preparation:

- Tokenize the text data (convert text into sequences of words or subwords). You can use nltk toolkit or torchtext.data.utils.get_tokenizer. Experiment with at least two different tokenization methods (e.g., nltk.word_tokenize vs. get_tokenizer("basic_english")). Justify your final choice. Compare the vocabulary sizes and processing times.
- Build a vocabulary from the training data only. Map each unique token to an integer index. You can use torchtext.vocab.build_vocab_from_iterator or create your own vocabulary mapping. Convert text data to numerical sequences using vocabulary.
- Pad sequences to a uniform length. Choose an appropriate maximum sequence length (max_length). You can base this on the distribution of review lengths (e.g., the 95th percentile).
- Use torch.nn.utils.rnn.pad_sequence (preferred) or manually pad with zeros.
- Discuss the impact of padding and truncation on model performance. If you truncate very long sequences, you might lose important information. If you pad excessively, you might introduce noise.

8. Split dataset into train, validation, and test sets. Because this is not a time series task, you can use train_test_split

Step 2: Baseline LSTM Model

1. Build an LSTM model with the following characteristics:
   - Embedding layer to represent words as dense vectors. Experiment with different embedding dimensions and explain your choice.
   - Include at least three LSTM layers (torch.nn.LSTM). Experiment with the number of hidden units in each layer.
   - Use dropout (torch.nn.Dropout) after *each* LSTM layer for regularization.
   - Include one or more fully connected layers (torch.nn.Linear) to map the LSTM output to the number of sentiment classes.
   - Use the output layer with an appropriate activation function (e.g., softmax for multi-class classification, sigmoid for binary classification).
   - Print model summary using torchinfo.summary
2. Train your model:
   - Choose an appropriate loss function (e.g., cross-entropy) and optimizer (e.g., Adam).
   - Train your model and monitor its performance on the training and validation sets.
   - Tune hyperparameters (learning rate, batch size, number of epochs, hidden units, dropout rate) using the validation set. Describe your tuning strategy (e.g., grid search, random search, manual tuning) and document the different

combinations you tried.
- Plot the training and validation loss and accuracy curves over epochs. Analyze the plots for signs of overfitting or underfitting. Adjust your model or hyperparameters if needed.

3. Evaluation and analysis:
   - Evaluate your best model on the test set. Report the accuracy, loss, and any other relevant metrics (precision, recall, F1-score, etc.).
   - Generate a confusion matrix (using sklearn.metrics.confusion_matrix and seaborn.heatmap). Analyze the confusion matrix.
4. Save the weights of the trained neural network that provides the best results. Check saving and loading models (PyTorch)

Step 3: Improved LSTM Model

1. Improve your baseline LSTM model by incorporating one or more of the following techniques:
   - Bidirectional LSTM. Use bidirectional=True in your LSTM layer.
   - Stacked LSTM. You already have at least three layers; you can add more if desired.
   - GRU (Gated Recurrent Unit):.Replace LSTM layers with torch.nn.GRU.
   - Attention Mechanism. Implement an attention mechanism. This is more advanced.
   - Pre-trained Word Embeddings. Initialize your embedding layer with pre-trained word embeddings (e.g., GloVe, Word2Vec, FastText). You can download pre-trained embeddings from various sources (e.g., the GloVe website). Use torchtext.vocab.Vectors to load pre-trained embeddings. This is highly recommended.
2. Create a new class for your improved model
3. Follow the same training and evaluation procedures as in step 2 (baseline LSTM model). Tune hyperparameters and analyze the performance on the training, validation, and test sets.
4. Directly compare the performance of your improved model to the baseline model. Use the same metrics and create plots that show both models' performance (e.g., training/validation curves on the same plot). Discuss the reasons for any observed improvements (or lack thereof). Analyze the impact of the specific techniques you used.
5. Save the weights of the trained neural network that provides the best results. Check saving and loading models (PyTorch)

Step 4: Discussion

1. Briefly describe the dataset and its key characteristics.

2. Describe both your baseline and improved LSTM architectures in detail.
3. Discuss the results, comparing the performance of the baseline and improved models. Refer to specific metrics and visualizations.
4. Discuss the strengths and limitations of using recurrent neural models (specifically LSTMs) for sentiment analysis. Consider aspects like:
   - Handling long sequences.
   - Computational cost.
   - Interpretability (compared to simpler models like logistic regression).
   - Sensitivity to hyperparameters.
5. **References.** Include details on all the resources used to complete this part.

Part V: CNN & LSTM Theoretical Part [10 points]

This part focuses on the theoretical analysis of CNNs and LSTMs through problem-solving. Provide clear and concise solutions, demonstrating your complete work and reasoning. Answers may be handwritten (legibly) or prepared using MS Word or LaTeX.

Part V.I: CNN [5 points]

You are working on a computer vision project using a CNN to classify images of human faces expressing different emotions. This is a multi-class classification task with 5 output classes: happiness, sadness, anger, surprise, and neutral.

CNN Architecture:

- Input: 32x28 RGB images.
- First Layer: Convolutional layer with 10 filters, each of size 5x5, zero padding, and a stride of 1.
- Output: Predicted probabilities for each category (5 classes), summing to 1.

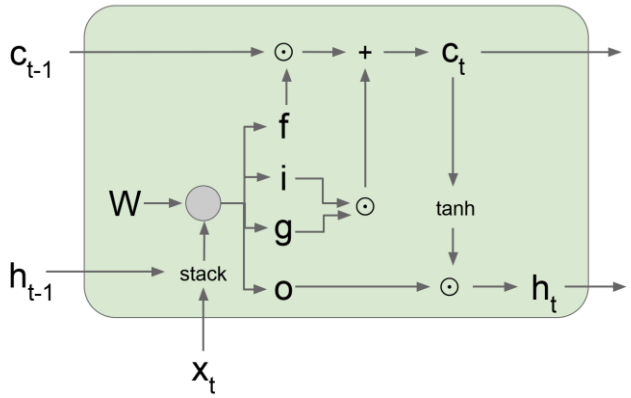
TASKS:

1. Calculate the output size (dimensions) after the first convolutional layer. Show your calculations.
2. Determine the total number of learnable parameters in the first convolutional layer. Show your calculations.
3. Recalculate the output size after the first convolutional layer if a padding of 1 was used instead of zero padding. Show your calculations.
4. Calculate the number of parameters in the first layer if the input images were grayscale (single channel) instead of RGB. Show your calculations.
5. Given the task and the requirement of probabilistic outputs, which activation function is most suitable for the output layer? Explain your choice and why other common activation functions are less appropriate in this scenario.

6. Prove that the activation function you chose in Task 5 is invariant to constant shifts in the input values. Mathematically demonstrate that adding a constant value to all input values will not change the resulting output probabilities. Explain the significance of shift invariance in this context.

Part V.II: LSTM Derivation [5 points]

Consider the following standard LSTM structure



i input gate: whether to write to cell

f forget gate: whether to erase cell

o output gate: how much to reveal cell

g gate gate: how much to write to cell

The definitions are provided as follows:

$$\begin{pmatrix} \mathbf{i}_t \\ \mathbf{f}_t \\ \mathbf{o}_t \\ \mathbf{g}_t \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} \left[\begin{pmatrix} \mathbf{W}_1 \\ \mathbf{W}_2 \\ \mathbf{W}_3 \\ \mathbf{W}_4 \end{pmatrix} \begin{pmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{pmatrix} \right]$$

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{g}_i$$

$$\mathbf{h}_t = \boldsymbol{o}_t \odot \tanh(\mathbf{c}_t)$$

where:

- σ denotes the sigmoid activation function
- $\mathbf{W}_1, \mathbf{W}_2, \mathbf{W}_3, \mathbf{W}_4$ represent weight matrices
- $\mathbf{h}_{t-1}$ represents the hidden state at previous time step
- $\mathbf{x}_t$ represents the current input.

TASKS:

Assume $\mathcal{L}$ is the loss of an LSTM model at time step *t*. Derive the gradient formulas for the following partial derivatives, expressing them in terms of the LSTM variables:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{i}_t}, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{f}_t}, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{o}_t}, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{g}_t}, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{c}_t}, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t}$$

Submission of theoretical part (Part V):

Your final pdf for Part V should contain clear, step-by-step derivations for all the requested

calculations. Justify each step and clearly mark your final answers.

You may provide the answer using handwritten notes, typed in the MS Word or Latex. For the final submission, combine all your answers into a single PDF named:

a1_part_v_TEAMMATE1_TEAMMATE2.pdf

e.g. a1_part_v_avereshc_mjadhav.pdf

Bonus points [max 10 points]

ResNeXt [7 points]

Based on your ResNet implementation in Part I, extend it to ResNeXT. It is expected that your accuracy is higher than ResNet. Compare the results with your VGG and ResNet implementation.

STEPS:

1. Implement the ResNeXT architecture. Pay close attention to the grouped convolutions and cardinality parameter. Using inbuild ResNeXt model won't be considered for evaluation.
2. Train and evaluate your ResNeXt model on the same dataset used in Part I.
3. Compare the performance of your ResNeXt model against your previous ResNet and VGG models. Provide a detailed analysis of the results, including:
    a. A table summarizing the performance metrics (accuracy, loss, etc.) for all three models.
    b. Discussion of the observed differences in performance. Explain why ResNeXt might be outperforming ResNet and VGG. Consider factors like cardinality, grouped convolutions, and the overall architecture.
    c. Analysis of any challenges encountered during the implementation or training process.
    d. Provide detailed analysis of the results.
4. **References.** Include details on all the resources used to complete this part.

SUBMISSION:

Submit your ResNeXt implementation as a Jupyter Notebook file named:

a1_bonus_resnext_TEAMMATE1_TEAMMATE2.ipynb

Ensure your notebook includes clear comments explaining your code and analysis.

Transfer Learning with Pre-trained Models [3 points]

Use in-build models with pre-trained weights and apply them to the Food-11 dataset.

STEPS:

1. Selected at least THREE different pre-trained models, e.g. ShuffleNet, Inception V3, and MobileNet V3. Check PyTorch documentation for more details. Justify your choice of models, considering their architectural strengths and suitability for the task.
2. For each chosen model:
    a. Load the pre-trained model and modify the classification head (the final fully connected layer) to match the number of classes in the Food-11 dataset.
    b. Fine-tune the model. Experiment with different hyperparameter settings (learning rate, batch size, etc.) to optimize performance. Explain your tuning strategy.
    c. Evaluate the performance of each fine-tuned model on the Food-11 dataset.
    d. Compare the results obtained with the different pre-trained models. Discuss which model performed best and analyze the reasons for the observed differences in performance.
3. **References.** Include details on all the resources used to complete this part.

SUBMISSION:

Submit your implementation as a Jupyter Notebook file named:

a1_bonus_transfer_learning_TEAMMATE1_TEAMMATE2.ipynb

Ensure your notebook includes clear comments explaining your code and analysis.

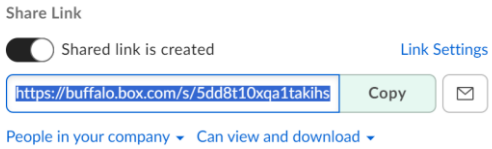
ASSIGNMENT STEPS

1. Submit checkpoint: Part I, Part II & Part V.I (February 25)

- Complete Part I, Part II & Part V.I: CNN of the assignment.
- Include all the references at the end of each part.
- Your Jupyter notebook should be saved with the outputs.
- Saved weights of the trained networks:
    - Go to UBbox > New > Folder > CSE 676-B Assignment 1 by TEAMMATE 1 & TEAMMATE 2
    - Upload your saved weights into this folder. Include the model weights that generate the best results for your model, named as a1_part#_TEAMMATE1_ TEAMMATE2.pt

      e.g. a1_part1_avereshc_ manasira.pt

- Copy a shared link to the UBbox folder, so it can be viewed by people in your company & it can be viewed and downloaded.



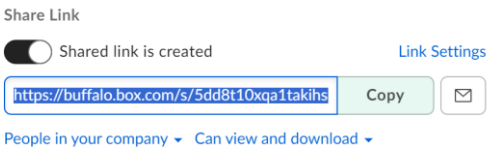
- Add the link to the txt file, named as a1_weights_TEAMMATE1_ TEAMMATE2.txt
- Submit this txt file as part of your submission on UBlearns

- Combine all files in a single zip folder that will be submitted on UBlearns, named as assignmen1_checkpoint_YOUR_UBIT.zip
- Submit to UBLearns > Assignments
- Suggested file structure:

  assignment1_checkpoint_TEAMMATE1_ TEAMMATE1.zip
  - a1_part_1_ TEAMMATE1_ TEAMMATE1.ipynb
  - a1_part_2_ TEAMMATE1_ TEAMMATE1.ipynb
  - a1_part_5_ TEAMMATE1_ TEAMMATE1.pdf
  - a1_weights_TEAMMATE1_ TEAMMATE2.txt
  - a1_datasets_TEAMMATE1_ TEAMMATE2.txt

2. Submit final results (March 6)

- Fully complete all parts of the assignment.
- Add all your assignment files in a zip folder including ipynb files for Part I, Part II, Part III, Part IV, Part V & Bonus part (optional) and txt file with a link to UBbox with links to weights and datasets.
- Datasets:
  - Go to UBbox > CSE 676-B Assignment 1 by TEAMMATE 1 & TEAMMATE 2
  - Upload your chosen datasets into this folder that you used for this assignment. There is no need to upload CNN dataset, only those that you've selected for Part III & Part IV
  - Copy a shared link to the UBbox folder, so it can be viewed by people in your company & it can be viewed and downloaded.



  - Add the link to the txt file, named as a1_datasets_TEAMMATE1_

TEAMMATE2.txt

- Submit this txt file as part of your submission on UBlearns

- Name zip folder with all the files as assignment1_final_ TEAMMATE1_ TEAMMATE2.zip

e.g. assignment1_final_avereshc_manasira.zip

- Submit to UBLearns > Assignments
- Your Jupyter notebook should be saved with the outputs.
- Include all the references at the end of each part that have been used to complete that part.
- You can make unlimited number of submissions and only the latest will be evaluated
- Suggested file structure (bonus part is optional):

assignment1_final_TEAMMATE1_ TEAMMATE1.zip

- a1_part_1_ TEAMMATE1_ TEAMMATE1.ipynb
- a1_part_2_ TEAMMATE1_ TEAMMATE1.ipynb
- a1_part_3_ TEAMMATE1_ TEAMMATE1.ipynb
- a1_part_4_ TEAMMATE1_ TEAMMATE1.ipynb
- a1_part_5_ TEAMMATE1_ TEAMMATE1.pdf
- a1_weights_TEAMMATE1_ TEAMMATE2.txt
- a1_datasets_TEAMMATE1_ TEAMMATE2.txt
- a1_bonus_ resnext_TEAMMATE1_ TEAMMATE1.ipynb
- a1_bonus_ transfer_learning_TEAMMATE1_ TEAMMATE1.ipynb

Notes:

- Ensure your code is well-organized and includes comments explaining key functions and attributes. After running the Jupyter Notebooks, all results and plots used in your report should be generated and clearly displayed.
- You can submit multiple files, but they all need to be labeled with a clear name.
- Recheck the submitted files, e.g. download and open them, once submitted and verify that they open correctly
- Large files: any individual file that is larger then 10MB (e.g. your model weights or datasets) should be uploaded to UBbox and you have to provide a link to them. A penalty of -10pts will be applied towards the assignment if there is a file submitted that is bigger than 10MB.
- It's ok if your final combined zip folder is larger then 10MB.
- The zip folder should include all relevant files, clearly named as specified.
- Only files uploaded on UBlearns and a submitted link to UBbox for saved weights are considered for evaluation.

Academic Integrity

The standing policy of the Department is that all students involved in any academic integrity violation (e.g., plagiarism in any way, shape, or form) will receive an F grade for the course. The catalog describes plagiarism as “Copying or receiving material from any source and submitting that material as one’s own, without acknowledging and citing the particular debts to the source, or in any other manner representing the work of another as one’s own.”. Refer to the Office of Academic Integrity for more details.

Important Information

This assignment can be done individually or in a team of up to two students. The grading rubrics are the same for a team of any size.

- No collaboration, cheating, and plagiarism is allowed in assignments, quizzes, the midterms or final project.
- All the submissions will be checked using MOSS as well as other tools. MOSS is based on the submitted works for the past semesters as well the current submissions. We can see all the sources, so you don't need to worry if there is a high similarity with your Checkpoint submission.
- The submissions should include all the references. Kindly note that referencing the source does not mean you can copy/paste it fully and submit as your original work. Updating the hyperparameters or modifying the existing code is a subject to plagiarism. Your work has to be original. If you have any doubts, send a private post on piazza to confirm.
- All parties involved in any suspicious cases will be officially reported using the Academic Dishonesty Report form. Please refer to the Academic Integrity Policy for more details.

Late Days Policy

You can use up to 5 late days throughout the course that can be applied to any assignment-related due dates. You do not have to inform the instructor, as the late submission will be tracked in UBlearns.

Important Dates

February 25, Tuesday, 11:59 pm - Checkpoint Submission is Due

March 6, Thu, 11:59 pm - Final Submission is Due