Homework #4

CSE 446/546: Machine Learning
Prof. Kevin Jamieson, Jamie Morgenstern
Due: **Wednesday** 12/16/2020 11:59 PM

Please review all homework guidance posted on the website before submitting to Gradescope. Reminders:

- Please provide succinct answers along with succinct reasoning for all your answers. Points may be deducted if long answers demonstrate a lack of clarity. Similarly, when discussing the experimental results, concisely create tables and/or figures when appropriate to organize the experimental results. In other words, all your explanations, tables, and figures for any particular part of a question must be grouped together, which also includes **your code and plots**.
- When submitting to gradescope, please link each question from the homework in gradescope to the location of its answer in your homework PDF. Failure to do so may result in point deductions. For instructions, see `https://www.gradescope.com/get_started#student-submission`.
- Please recall that B problems, indicated in boxed text are only graded for 546 students, and that they will be weighted at most 0.2 of your final GPA (see website for details). In Gradescope there is a place to submit solutions to A and B problems separately. You are welcome to create just a single PDF that contains answers to both, submit the same PDF twice, but associate the answers with the individual questions in gradescope.
- If you collaborate on this homework with others, you must indicate who you worked with on your homework. Failure to do so may result in accusations of plagiarism.

In the 'A' problems on this assignment, you are provided some choice about which subset of problems to complete. A subset of the problems are color-coded. You may choose **one BLUE problem from {A5, A6 }** to complete. You do not need to complete both problems of the same color. Problems of the same color have the same total point value. Problems {A1, A2, A3, A4, A7} are not optional and should be completed by all students.

Conceptual Questions

A1. The answers to these questions should be answerable without referring to external materials. Briefly justify your answers with a few words.

a. *[2 points]* True or False: Training deep neural networks requires minimizing a non-convex loss function, and therefore gradient descent might not reach the globally-optimal solution.

b. *[2 points]* True or False: It is a good practice to initialize all weights to zero when training a deep neural network.

c. *[2 points]* True or False: We use non-linear activation functions in a neural network's hidden layers so that the network learns non-linear decision boundaries.

d. *[2 points]* True or False: Given a neural network, the time complexity of the backward pass step in the backpropagation algorithm can be prohibitively larger compared to the relatively low time complexity of the forward pass step.

e. *[2 points]* True or False: Autoencoders, where the encoder and decoder functions are both neural networks with nonlinear activations, can capture more variance of the data in its encoded representation than PCA using the same number of dimensions.

Unsupervised Learning with Autoencoders

A2. In this exercise, we will train two simple autoencoders to perform dimensionality reduction on MNIST. As discussed in lecture, autoencoders are a long-studied neural network architecture comprised of an encoder component to summarize the latent features of input data and a decoder component to try and reconstruct the original data from the latent features.

Weight Initialization and PyTorch

We recommend using `nn.Linear` for your linear layers. You will not need to initialize the weights yourself; the default He/Kaiming uniform initialization in PyTorch will be sufficient. *Hint: we also recommend using the* ***`nn.Sequential`*** *module to organize your network class and simplify the process of writing the forward pass.*

Training

Use `optim.Adam` for this question. Experiment with different learning rates. Use mean squared error (`nn.MSELoss()` or `F.mse_loss()`) for the loss function and ReLU for the non-linearity in b.

a. *[10 points]* Use a network with a single linear layer. Let $W_{\mathrm{e}} \in \mathbb{R}^{h \times d}$ and $W_{\mathrm{d}} \in \mathbb{R}^{d \times h}$. Given some $x \in \mathbb{R}^d$, the forward pass is formulated as

$$\mathcal{F}_1(x) = W_{\mathrm{d}} W_{\mathrm{e}} x.$$

Run experiments for $h \in \{32, 64, 128\}$. For each of the different h values, report your final error and visualize a set of 10 reconstructed digits, side-by-side with the original image. *Note:* we omit the bias term in the formulation for notational convenience since `nn.Linear` learns bias parameters alongside weight parameters by default.

b. *[10 points]* Use a single-layer network with non-linearity. Let $W_{\mathrm{e}} \in \mathbb{R}^{h \times d}$, $W_{\mathrm{d}} \in \mathbb{R}^{d \times h}$, and activation $\sigma : \mathbb{R} \longmapsto \mathbb{R}$. Given some $x \in \mathbb{R}^d$, the forward pass is formulated as

$$\mathcal{F}_2(x) = \sigma(W_{\mathrm{d}} \sigma(W_{\mathrm{e}} x)).$$

Report the same findings as asked for in part a (for $h \in \{32, 64, 128\}$.

c. *[5 points]* Now, evaluate $\mathcal{F}_1(x)$ and $\mathcal{F}_2(x)$ (use $h = 128$ here) on the test set. Provide the test reconstruction errors in a table.

d. *[5 points]* In a few sentences, compare the quality of the reconstructions from these two autoencoders compare with those of PCA from last assignment. You may want to re-run your code for PCA using the different h values as the number of top-k eigenvalues.

Using Pretrained Networks and Transfer Learning

A3. In this problem, instead of training a neural network from scratch, we will use one that has already been trained on a very large dataset (ImageNet) and adjust it for the task at hand. This process of adapting weights in a model trained for another task is known as *transfer learning*.

- Begin with the pre-trained AlexNet[1] model from `torchvision.models` for both tasks below.
- Do not modify any module within AlexNet that is not the final classifier layer.
- The output of AlexNet comes from the 6th layer of the classifier. Specifically, `model.classifer[6] = nn.Linear(4096, 1000)`. To use AlexNet with CIFAR-10, we will reinitialize (replace) this layer with `nn.Linear(4096, 10)`. This re-initializes the weights, and changes the output shape to reflect the desired number of target classes in CIFAR-10.

[1] AlexNet achieved an early breakthrough performance on `ImageNet` and was instrumental in sparking the deep learning revolution in 2012.

We will explore two different ways to formulate transfer learning.

a. *[15 points]* **Use AlexNet as a fixed feature extractor:** Add a new linear layer to replace the existing classification layer, and adjust only the weights of this new layer (keeping the weights of all other layers fixed). Provide plots for training loss and validation loss over the number of epochs. Report the highest validation accuracy achieved. Finally, evaluate the model on the test data and report both the accuracy and the loss.

   When using AlexNet as a fixed feature extractor, make sure to freeze all of the parameters in the network *before* adding your new linear layer:

```
model = torchvision.models.alexnet(pretrained=True)
for param in model.parameters():
    param.requires_grad = False
model.classifier[6] = nn.Linear(4096, 10)
```

b. *[15 points]* **Fine-Tuning:** The second approach to transfer learning is to fine-tune the weights of the pre-trained network, in addition to training the new classification layer. In this approach, all network weights are updated at every training iteration; we simply use the existing AlexNet weights as the "initialization" for our network (except for the weights in the new classification layer, which will be initialized using whichever method is specified in the constructor) prior to training on CIFAR-10. Following the same procedure, report all the same metrics and plots as in the previous question. (Note: fine-tune AlexNet on CPU takes an insame amount of time, so we recommend you to use Google Colab, which has free GPU access. To enable GPU for the notebook: Navigate to Edit→Notebook Settings. select GPU from the Hardware Accelerator drop-down. For information about training on GPU, click on the following link: Training on GPU)

Image Classification on CIFAR-10

A4. In this problem we will explore different deep learning architectures for image classification on the CIFAR-10 dataset. If you are not comfortable with PyTorch from the previous lecture and discussion materials, use the tutorials at `http://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html` and make sure you are familiar with tensors, two-dimensional convolutions (`nn.Conv2d`) and fully-connected layers (`nn.Linear`), ReLU non-linearities (`F.relu`), pooling (`nn.MaxPool2d`), and tensor reshaping (`view`).

For this problem, it is highly recommended that you copy and modify the existing network code produced in the tutorial *Training a classifier*. You should not be coding this network from scratch!

A few preliminaries:

- Each network f maps an image $x^{in} \in \mathbb{R}^{32\times32\times3}$ (3 channels for RGB) to an output $f(x^{in}) = x^{out} \in \mathbb{R}^{10}$. The class label is predicted as $\arg\max_{i=0,1,\ldots,9} x_i^{out}$.
- The network is trained via multiclass cross-entropy loss.
- Create a validation dataset by appropriately partitioning the train dataset. *Hint*: look at the documentation for `torch.utils.data.random_split`. Make sure to tune hyperparameters like network architecture and step size on the validation dataset. Do **NOT** validate your hyperparameters on the test dataset.
- Modify the training code such that at the end of each epoch (one pass over the training data) it computes and prints the training and validation classification accuracy.
- While one would usually train a network for hundreds of epochs to reach convergence and maximize accuracy, this can be prohibitively time-consuming, so feel free to train for just a dozen or so epochs.

For all of the following, apply a hyperparameter tuning method (grid search, random search, etc.) using the validation set, report the hyperparameter configurations you evaluated and the best set of hyperparameters from this set, and plot the training and validation classification accuracy as a function of iteration. Produce a separate line or plot for each hyperparameter configuration evaluated. Finally, evaluate your best set of hyperparameters on the test data and report the accuracy. **On the larger networks, you should expect to tune hyperparameters and train to at least** 70% **accuracy.**

Here are the network architectures you will construct and compare.

a. *[15 points]* **Fully-connected output, 0 hidden layers (logistic regression):** this network has no hidden layers and linearly maps the input layer to the output layer. This can be written as

$$x^{out} = W\text{vec}(x^{in}) + b$$

where $x^{out} \in \mathbb{R}^{10}$, $x^{in} \in \mathbb{R}^{32\times32\times3}$, $W \in \mathbb{R}^{10\times3072}$, $b \in \mathbb{R}^{10}$ since $3072 = 32 \cdot 32 \cdot 3$. For a tensor $x \in \mathbb{R}^{a\times b\times c}$, we let $\text{vec}(x) \in \mathbb{R}^{abc}$ be the reshaped form of the tensor into a vector (in an arbitrary but consistent pattern).

b. *[15 points]* **Fully-connected output, 1 fully-connected hidden layer:** this network has one hidden layer denoted as $x^{hidden} \in \mathbb{R}^M$ where M will be a hyperparameter you choose (M could be in the hundreds). The nonlinearity applied to the hidden layer will be the `relu` ($\text{relu}(x) = \max\{0, x\}$. This network can be written as

$$x^{out} = W_2\text{relu}(W_1\text{vec}(x^{in}) + b_1) + b_2$$

where $W_1 \in \mathbb{R}^{M\times3072}$, $b_1 \in \mathbb{R}^M$, $W_2 \in \mathbb{R}^{10\times M}$, $b_2 \in \mathbb{R}^{10}$.

c. *[15 points]* **Convolutional layer with max-pool and fully-connected output:** for a convolutional layer W_1 with filters of size $k \times k \times 3$, and M filters (reasonable choices are $M = 100$, $k = 5$), we have that $\text{Conv2d}(x^{in}, W_1) \in \mathbb{R}^{(33-k)\times(33-k)\times M}$.

- Each convolution will have its own offset applied to each of the output pixels of the convolution; we denote this as $\text{Conv2d}(x^{in}, W) + b_1$ where b_1 is parameterized in $\mathbb{R}^M$. Apply a `relu` activation to the result of the convolutional layer.
- Next, use a max-pool of size $N \times N$ (a reasonable choice is $N = 14$ to pool to 2×2 with $k = 5$) we have that $\text{MaxPool}(\text{relu}(\text{Conv2d}(x^{in}, W_1) + b_1)) \in \mathbb{R}^{\lfloor \frac{33-k}{N} \rfloor \times \lfloor \frac{33-k}{N} \rfloor \times M}$.
- We will then apply a fully-connected layer to the output to get a final network given as

$$x^{output} = W_2 \text{vec}(\text{MaxPool}(\text{relu}(\text{Conv2d}(x^{input}, W_1) + b_1))) + b_2$$

  where $W_2 \in \mathbb{R}^{10 \times M(\lfloor \frac{33-k}{N} \rfloor)^2}$, $b_2 \in \mathbb{R}^{10}$.

The parameters M, k, N (in addition to the step size and momentum) are all hyperparameters, but you can choose a reasonable value.

d. *[5 points]* **Tuning:** Return to the original network you were left with at the end of the tutorial *Training a classifier*. (Note that this is not the network from part (c) above.) Tune the different hyperparameters (number of convolutional filters, filter sizes, dimensionality of the fully-connected layers, stepsize, etc.) and train for a sufficient number of iterations to achieve a *train accuracy* of at least 87%. You may not modify the core structure of the model (i.e., adding additional layers). Provide the hyperparameter configuration used to achieve this performance. Make sure to **save this model** so that you can do the next part (see the *Training a classifier* tutorial for details on how to do this).

e. *[10 points]* **Adversarial Attacks:** Modern deep neural networks are brittle and susceptible to small perturbations to their inputs. This gives rise to *adversarial examples*, which are nearly indistinguishable to the human eye but somehow "fool" neural networks into making drastically wrong predictions.

One algorithm to generate such examples is the untargeted fast gradient sign method (FGSM) attack, which can be described as follows: let x be an input image with label y, $\mathcal{F}$ be a neural network, and ϵ be a small value (intuitively, an attack rate).

$$\begin{aligned} \hat{y} &= \mathcal{F}(x) \\ \mathcal{L} &= \text{CrossEntropy}(\hat{y}, y) \\ x' &= x + \epsilon * \text{sign}\left(\nabla_x \mathcal{L}\right) \end{aligned}$$

where $\text{sign}(t) = \frac{t}{|t|}$. We then use x' as an input to the network. Note that the calculation for x' loosely resembles gradient descent. Intuitively, we are slightly adjusting the input image so that the model is less likely to predict its true class.

For this part, use your classifier from part (d) to do the following steps. As always, please provide all code and plots.

1. Select four images from the train set that have been correctly classified. Visualize them and provide their labels.
2. Implement the untargeted FGSM algorithm. Run one iteration on these images and visualize them: they should look like the originals.
3. Provide the predicted labels for your attacked images. You should have at least one image that is incorrectly classified. **Remark:** FGSM is a simple attack, but it's not always effective. In order to generate successful adversarial examples, you may need to try different values of ϵ or even different images, depending on where your classifier excels.
4. Explain the significance of the existence of such adversarial examples.

The number of hyperparameters to tune in exercise (d), combined with the slow training times, will hopefully give you a taste of how difficult it is to construct networks with good generalization performance. State-of-the-art networks can have dozens of layers, each with their own hyperparameters to tune. Additional hyperparameters you are welcome to play with if you are so inclined, include: changing the activation function, replace max-pool with average-pool, and experimenting with batch normalization or dropout.

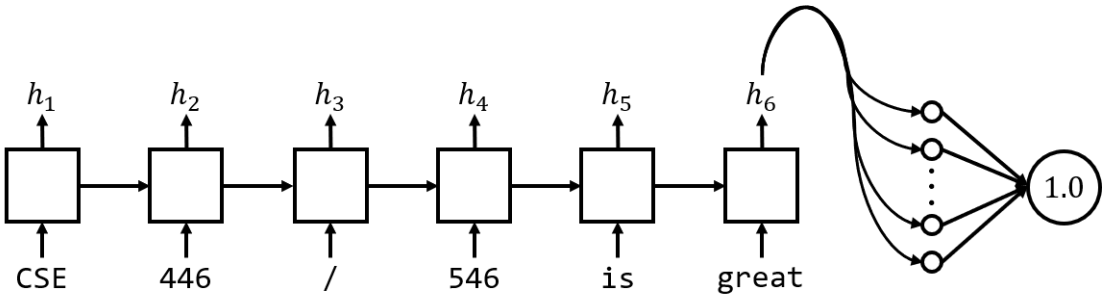
Text classification on SST-2

A5. The Stanford Sentiment Treebank (SST-2) is a dataset of movie reviews. Each review is annotated with a label indicating whether the sentiment of the review is positive or negative. Below are some examples from the dataset. Note that often times the reviews are only partial sentences or even single words.

Sequence	Sentiment
is one big , dumb action movie .	Negative
perfectly executed and wonderfully sympathetic characters ,	Positive
until then there 's always these rehashes to feed to the younger generations	Negative
is like nothing we westerners have seen before .	Positive

In this problem you will use a Recurrent Neural Network (RNN) for classifying reviews as either Positive (1) or Negative (0).

Using an RNN for binary classification



Above is a simplified visualization of the RNN you will be building. Each token of the input sequence (*CSE*, *446*, ...) is fed into the network sequentially. Note that in reality, we convert each token to some integer index. But training with discrete values does not work well, so we also "embed" the words in a high-dimensional continuous space. We have already provided an `nn.Embedding` layer to you to do this. Each RNN cell (squares above) generates a hidden state h_i. We then feed the last hidden state into a simple fully-connected layer, which then produces a single prediction between 0 and 1.

Setup

1. Download the GloVe embeddings from `http://nlp.stanford.edu/data/glove.6B.zip`. Extract the zip file and move the file `glove.6B.50d.txt` into the same folder as the starter code (`util.py, train.py, hw4_a5.py`).
2. Download the SST-2 dataset from here `https://gluebenchmark.com/tasks` (Click on download next to `The Stanford Sentiment Treebank`). Extract the zip file into the same folder as above.

You only need to modify `hw4_a5.py`, however you are free to also modify the other two files. You will only submit `hw4_a5.py`, but if you make changes to the other files you should also include them in your submission.

Problems

a. *[10 points]* In Natural Language Processing (NLP), we usually have to deal with variable length data. In SST-2, each data point corresponds to a review, and reviews often have different lengths. The issue is that to efficiently train a model with textual data, we need to create batches of data that are fixed size matrices. In order to store a batch of sequences in a single matrix, we add padding to shorter sequences so that each sequence has the same length. Given a list of N sequences, we:

   1. Find the length of the longest sequence in the batch, call it `max_sequence_length`

2. Append padding tokens to the end of each sequence so that each sequence has length `max_sequence_length`
3. Stack the sequences into a matrix of size `(N, max_sequence_length)`

In this process, words are mapped to integer ids, so in the above process we actually store integers rather than strings. For the padding token, we simply use id 0. In the file `hw4_a5.py`, fill out `collate_fn` to perform the above batching process. Details are provided in the comment of the function.

b. *[15 points]* Implement the constructor and forward method for `RNNBinaryClassificationModel`. You will use three different types of recurrent neural networks: the vanilla RNN (`nn.RNN`), Long Short-Term Memory (`nn.LSTM`) and Gated Recurrent Units (`nn.GRU`). For the hidden size, use 64 (Usually this is a hyperparameter you need to tune). We have already provided you with the embedding layer to turn token indices into continuous vectors of size 50, but you will need a linear layer to transform the last hidden state of the recurrent layer to a shape that can be interpreted as a label prediction. The code for each of the three networks will only differ by the recurrent layer you use. In your code submission, use any of the three layers.

c. *[5 points]* Implement the `method` of `RNNBinaryClassificationModel`, which should compute the binary cross-entropy loss between the predictions and the target labels. Also implement the `accuracy` method, which given the predictions and the target labels should return the accuracy.

d. *[15 points]* We have already provided all of the data loading, training and validation code for you. Choose an appropriate batch size, learning rate and number of epochs and set the constants `TRAINING_BATCH_SIZE`, `NUM_EPOCHS`, `LEARNING_RATE` accordingly. With a good learning rate, you shouldn't have to train for more than 16 epochs. Report your best validation loss and corresponding validation accuracy, corresponding training loss and training accuracy for each of the three types of recurrent neural networks.

e. *[5 points]* Our recurrent neural network processes the text input from left to right. What assumptions is this making about the structure of the input, and what are some potential problems with this setup?

f. *[5 points]* Our model passes the final hidden state of our recurrent neural network into the classification layer. In a few sentences, answer the following questions: Why does this make sense? What are some potential problems with this setup?

g. *[5 points]* Currently, we are only using the final hidden state of the RNN to classify the entire sequence as having either positive or negative sentiment. But can we make use of the other hidden states? Suppose you wanted to use the same architecture for a related task called tagging. For tagging, the goal is to predict a tag for each token of the sequence. For instance, we might want predict the part of speech tag (verb, noun, adjective etc.) of each token. In a few sentences, describe how you would modify the current architecture to predict a tag for each token.

h. (Extra Credit: *[1 points]*) When you run training, the model will print out 8 random reviews. What is the funniest review you have encountered after running the code a few times?

Topic Modeling

A6. In this problem, we will use a feedforward neural network to study **Topic Modeling**: given a corpus of documents, the goal is to create an interpretable vector representation of a document. We define **"topic"** as **a learned probability distribution over words**. For example, one distribution captures how people might rank words for "sports" and another for "politics". Because of different probability distributions of words, each topic can be treated as a clustering over words. By observing the K most important words (ranked by probability) in one topic, different people can come up with different names for them.

Model/Algorithm Description As input, the document i is a vector $\boldsymbol{x}_i \in \mathbb{R}^V$; each index of the vector uniquely maps to a word in the provided `vocabulary.json`. The value at each index is the frequency of the word corresponding to that index. We initially ignore the representation of our document and assume each document is generated from a latent variable, $\boldsymbol{z}_i$. To model this data-generating process, we simultaneously learn two modules:

a. (See Problem a) An *encoder*, which maps from the observed text an approximate posterior $q(\boldsymbol{z}_i \mid \boldsymbol{x}_i)$, the distribution over the hidden variable $\boldsymbol{z}_i$ *after* we observe input $\boldsymbol{x}_i$. Intuitively, this means encoder is guessing what is the probability distribution for hidden variable $\boldsymbol{z}$ that is used to generate the document. We make the assumption that our posterior distribution is a normally distributed posterior, i.e.,

$$q(\boldsymbol{z}_i \mid \boldsymbol{x}_i) = \mathcal{N}(\boldsymbol{z}_i \mid f_\mu(\boldsymbol{x}_i), \text{diag}(f_\sigma(\boldsymbol{x}_i)))$$

b. (See Problem b) A *decoder*, which reconstructs the text from the latent representation (encoded by the encoder),

$$\boldsymbol{x}_i \sim p(\boldsymbol{x}_i \mid f_d(\boldsymbol{z}_i))$$

Using standard principles of variational inference, we derive a variational bound on the marginal *log-likelihood* of the observed data. This metric is typically used in unsupervised setting for training a model; the intuition of this is that if the decoder can "reconstruct" the input $\boldsymbol{x}_i$ (using $\boldsymbol{z}_i$ inferred by encoder) with higher and higher probability, this might indicate (not a guarantee) the model as a whole is getting better at modeling this data-generating process.
It is necessary to use a bound as a proxy to this loss function because the exact value of this *log-likelihood* is hard to compute. In our training, the algorithm will keep increase the lower bound, so that we know the model is getting better if someone figure out a way to calculate the *log-likelihood*. (See Problem c).

$$\log p(\boldsymbol{x}_i) \geq \mathbb{E}_{q(\boldsymbol{z}_i|\boldsymbol{x}_i)}[\log p(\boldsymbol{x}_i \mid \boldsymbol{z}_i)] - \text{KL}[q(\boldsymbol{z}_i \mid \boldsymbol{x}_i) \parallel p(\boldsymbol{z}_i)] = \mathcal{B}(\boldsymbol{x}_i)$$

Since the variance of $q(\boldsymbol{z}_i \mid \boldsymbol{x}_i)$ is a diagonal matrix, we can use a *reparameterization trick* to approximate the expectation with a single sample:

$$\mathcal{B}(\boldsymbol{x}_i) \approx \log p(\boldsymbol{x}_i \mid \boldsymbol{z}_i^{(s)}) - \text{KL}[q(\boldsymbol{z}_i \mid \boldsymbol{x}_i) \parallel p(\boldsymbol{z}_i)]$$

$$\boldsymbol{z}_i^{(s)} = f_\mu(\boldsymbol{x}_i) + f_\sigma(\boldsymbol{x}_i) \cdot \boldsymbol{\varepsilon}^{(s)},$$

where $\boldsymbol{\varepsilon}^{(s)} \sim \mathcal{N}(0, \boldsymbol{I})$ is sampled from an independent normal distribution, and $\cdot$ deonotes a element-wise multiplication. For simplicity, we assume the prior distribution — our guess about $\boldsymbol{z}_i$ *before* we see any input — $p(\boldsymbol{z}_i) = \mathcal{N}(0, \boldsymbol{I})$.

Training Training this model will use mini-batch gradient descent on the loss function in 6.

Quality of learned topic We use Normalized Pointwise Mutual Information (NPMI) as a metric to compare different topic models. It meausres the probability that two words collocate in an **external corpus** (in our case, the validation data). For each topic t — a probability distribution over words in vocabulary set, we collect the top ten most probable words and compute NPMI between all pairs (See Problem e for more details):

$$\text{NPMI}(t) = \sum_{i,j \leq 10; j \neq i} \frac{\log \frac{P(t_i, t_j)}{P(t_i), P(t_j)}}{-\log P(t_i, t_j)}, \tag{1}$$

where t_i denotes the ith word from your vocabulary set.
Note: For this question, feel free to use any PyTorch functionality. You are restricted to using PyTorch, Numpy, Scikit-learn and other non-deep-learning packages. If you found other package useful, please check with us before you use it. We already preprocess the data for you. Please see `README.md` from `ag.tgz`.

a. *[20 points]* From now on, we will use a single example for clear illustration. The encoder receive an input $\boldsymbol{x}_i$ and try to guess the corresponding $\boldsymbol{z}_i^{(s)}$. It has the following formulation:

$$\boldsymbol{h}_i = \text{ReLU}(\boldsymbol{W}_h \text{ReLU}(\boldsymbol{W}_x \boldsymbol{x}_i + \boldsymbol{b}_x) + \boldsymbol{b}_h) \quad \boldsymbol{W}_h \in \mathbb{R}^{h \times h}, \boldsymbol{W}_x \in \mathbb{R}^{h \times V}$$

$$\boldsymbol{\mu}_i = f_\mu(\boldsymbol{x}_i) = \boldsymbol{W}_\mu \boldsymbol{h}_i + \boldsymbol{b}_\mu, \boldsymbol{W}_\mu \in \mathbb{R}^{T \times h}$$

$$\boldsymbol{\sigma}_i = f_\sigma(\boldsymbol{x}_i) = \exp(\boldsymbol{W}_\sigma \boldsymbol{h}_i + \boldsymbol{b}_\sigma), \boldsymbol{W}_\sigma \in \mathbb{R}^{T \times h}$$

$$\boldsymbol{z}_i^{(s)} = \boldsymbol{\mu}_i + \boldsymbol{\sigma}_i \cdot \boldsymbol{\varepsilon}^{(s)}.$$

T denotes the number of topics used for this model. We recommend set $T \in [20, 100]$. It is a hyperparameter you can play with. Please implement this and include your code for encoder here. Note that $\cdot$ deonotes an element-wise multiplication. (Suggestion: Name your variables appropriately)

b. *[5 points]* The decoder has a simple formulation:

$$\boldsymbol{\theta}_i = \text{softmax}(\boldsymbol{z}_i^{(s)})$$

$$\boldsymbol{\eta}_i = \text{softmax}(\boldsymbol{B}\boldsymbol{\theta}_i), \boldsymbol{B} \in \mathbb{R}^{V \times T} \tag{2}$$

Implement this loss and include your code. (Suggestion: Name your variables appropriately). Note that decoder doesn't have an offset.

c. *[10 points]* For the decoder, we use the following form, which reconstruct the input in terms of topics (probability distributions over the vocabulary):

$$\log p(\boldsymbol{x}_i \mid \boldsymbol{z}_i^{(s)}) = \sum_{j=1}^{V} \boldsymbol{x}_{ij} \cdot \log \boldsymbol{\eta}_{ij},$$

where j ranges over the vocabulary.

Now we are done with the forward pass of the model. Let's focus on the backword pass by first looking at the lower bound $\mathcal{B}(\boldsymbol{x}_i)$:

$$\log p(\boldsymbol{x}_i \mid \boldsymbol{z}_i^{(s)}) - \text{KL}[q(\boldsymbol{z}_i \mid \boldsymbol{x}_i) \parallel p(\boldsymbol{z}_i)] \tag{3}$$

So, you can use the closed-form solution for KL here.[2] Because we are using gradient descent (i.e., decrease the loss), and yet we want to increase our lower bound, so our loss function is actually $-\mathcal{B}(\boldsymbol{x}_i)$. For mini-batch gradient descent, you just need to average the $-\mathcal{B}(\boldsymbol{x}_i)$s

Please implement this loss correctly and include your code here. (Suggestion: Name your variables appropriately).

d. *[5 points]* Wait a minute. If you train using the loss function above, you won't get good results! Likely, your T topics will be very similar to each other. Variational Autoencoder (VAE) is generally deemed as a mathematically appealing model, but this is still an open problem of how to train VAE properly. (For more information about VAE, click on the following link: VAE) We will introduce a scalar α before the KL term in Eq. 3, which we gradually increase from **0 to 1** (increase it every gradient update). There are many ways you can increase α, but the simplest increase is

$$\alpha \leftarrow \alpha + \frac{1}{\texttt{linear\_scaling}},$$

where `linear_scaling` should be at scale of 1000

Another option is

$$\alpha \leftarrow \alpha + \frac{1}{1 + \exp(-w_1 * (1 - w_2))},$$

recommended values are $w_1 = 0.25, w_2 = 15$. Please show us your rewritten loss function, and code related to updates. (Suggestion: Name your variables appropriately)

e. *[10 points]* The global NPMI is computed by averaging the NPMIs across all topics. For this problem, use the global NPMI(per epoch) as an *early stopping* criteria. **If NPMI doesn't increase for P epochs, we stop the training.** A recommended value for P is 5.

We will provide the code to help you calculate the NPMI score. You might want to make some modification to the provided code. Show your plot for NPMI v.s. training epoch. On the same plot, show loss v.s. training epoch as well.

f. *[10 points]* Now you trained your model. See further note on why this model is interpretable in the following section. For the purpose of this problem, you can get t th topic (i.e., a probability distribution) by simply setting $\boldsymbol{\theta}$ to be a one-hot vector where t th entry is 1 and 0 elsewhere. Then, use Eq. 2 to get it.

Choose 5 of your favorite topics, could you think of a good name for each topic? Then, show top 10 words from each.

[2] `https://en.wikipedia.org/wiki/Kullback%E2%80%93Leibler_divergence#Multivariate_normal_distributions`

Interpretability

By placing a softmax on $\boldsymbol{z}_i$, we can interpret $\boldsymbol{\theta}_i$ as a distribution over latent topics. In another word, each document can be represented as a "topic vector", which consists of different proportion of all the learned topics. See example here.
For example, $\boldsymbol{\theta}_i = [0.3, 0.7]$, and the learned topics can be labeled $[\text{Sport}, \text{News}]$. An intuitive understanding is "this document is 30% on sport and 70% on News."
Each topic could have the probability distribution over vocabulary set: Sport $= [0.4, 0.4, 0.2, 0.2]$ and News $= [0.1, 0.2, 0.4, 0.3]$, with a tiny (ordered) vocabulary set $= [\text{basketball}, \text{player}, \text{report}, \text{press}]$.
In this sense, $\boldsymbol{B}$ is the core of the topic model, as it contains positive and negative topical value before we multiply it with $\boldsymbol{\theta}$ and apply softmax.

Comparison with other methods (See lecture slide on text processing)

- Use tf-idf to calculate a feature vector for each document and run k-means on it. This method will be severely affected by the size of vocabulary set. The more words you consider, the more your algorithm will be affected by the "curse of dimensionality"
- Nonnegative matrix factorization. The downside is that matrix size is determined by the size of vocabulary set, which affect the compute time for factorization.

Matrix Completion and Recommendation System

B1. You will build a personalized movie recommender system. We will use the 100K MovieLens dataset available at `https://grouplens.org/datasets/movielens/100k/`. There are $m = 1682$ movies and $n = 943$ users. Each user rated at least 20 movies, but some watched many more. The total dataset contains $100,000$ total ratings from all users. The goal is to recommend movies the users haven't seen.
Consider a matrix $R \in \mathbb{R}^{m \times n}$ where the entry $R_{i,j} \in \{1, \ldots, 5\}$ represents the jth user's rating on movie i. A higher value represents that the user is more satisfied with the movie.

We may think of our historical data as some observed entries of this matrix while many remain unknown, and we wish to estimate the unknown ratings that each user would assign to each movie. We could use these ratings to recommend the "best" movies for each user.

The dataset contains user and movie metadata which we will ignore. We strictly use the ratings contained in the `u.data` file. Use this data file and the following python code to construct a training and test set:

```
import csv
import numpy as np
data = []
with open('u.data') as csvfile:
    spamreader = csv.reader(csvfile, delimiter='\t')
    for row in spamreader:
        data.append([int(row[0])-1, int(row[1])-1, int(row[2])])
data = np.array(data)

num_observations = len(data)   # num_observations = 100,000
num_users = max(data[:,0])+1   # num_users = 943, indexed 0,...,942
num_items = max(data[:,1])+1   # num_items = 1682 indexed 0,...,1681

np.random.seed(1)
num_train = int(0.8*num_observations)
perm = np.random.permutation(data.shape[0])
train = data[perm[0:num_train],:]
test = data[perm[num_train::],:]
```

The arrays `train` and `test` contain R_{train} and R_{test}, respectively. Each line takes the form "j, i, s", where j is the user index, i is the movie index, and s is the user's score.

Using `train`, you will train a model that can predict $\widehat{R} \in \mathbb{R}^{m \times n}$, how every user would rate every movie. You will evaluate your model based on the average squared error on `test`:

$$\mathcal{E}_{\text{test}}(\widehat{R}) = \frac{1}{|\text{test}|} \sum_{(i,j,R_{i,j}) \in \text{test}} (\widehat{R}_{i,j} - R_{i,j})^2.$$

Low-rank matrix factorization is a baseline method for personalized recommendation. It learns a vector representation $u_i \in \mathbb{R}^d$ for each movie and a vector representation $v_j \in \mathbb{R}^d$ for each user, such that the inner product $\langle u_i, v_j \rangle$ approximates the rating $R_{i,j}$. You will build a simple latent factor model.

You will implement multiple estimators and use the inner product $\langle u_i, v_j \rangle$ to predict if user j likes movie i in the test data. For simplicity, we will put aside best practices and choose hyperparameters by using those that minimize the test error. You may use fundamental operators from `numpy` or `pytorch` in this problem (`numpy.linalg.lstsq, SVD, autograd,` etc.) but not any precooked algorithm from a package

like `scikit-learn`. If there is a question whether some package is not allowed for use in this problem, it probably is not appropriate.

a. *[5 points]* Our first estimator pools all users together and, for each movie, outputs as its prediction the average user rating of that movie in `train`. That is, if $\mu \in \mathbb{R}^m$ is a vector where μ_i is the average rating of the users that rated the ith movie, write this estimator $\widehat{R}$ as a rank-one matrix. Compute the estimate $\widehat{R}$. What is $\mathcal{E}_{\text{test}}(\widehat{R})$ for this estimate?

b. *[5 points]* Allocate a matrix $\widetilde{R}_{i,j} \in \mathbb{R}^{m \times n}$ and set its entries equal to the known values in the training set, and 0 otherwise. Let $\widehat{R}^{(d)}$ be the best rank-d approximation (in terms of squared error) approximation to $\widetilde{R}$. This is equivalent to computing the singular value decomposition (SVD) and using the top d singular values. This learns a lower-dimensional vector representation for users and movies, assuming that each user would give a rating of 0 to any movie they have not reviewed.

- For each $d = 1, 2, 5, 10, 20, 50$, compute the estimator $\widehat{R}^{(d)}$. We recommend using an efficient solver such as `scipy.sparse.linalg.svds`.
- Plot the average squared error of predictions on the training set and test set on a single plot, as a function of d.

Note that, in most applications, we would not actually allocate a full $m \times n$ matrix. We do so here only because our data is relatively small and it is instructive.

c. *[10 points]* Replacing all missing values by a constant may impose strong and potentially incorrect assumptions on the unobserved entries of R. A more reasonable choice is to minimize the MSE (mean squared error) only on rated movies. Define a loss function:

$$L\Big(\{u_i\}_{i=1}^m, \{v_j\}_{j=1}^n\Big) := \sum_{(i,j,R_{i,j}) \in \text{train}} (\langle u_i, v_j \rangle - R_{i,j})^2 + \lambda \sum_{i=1}^m \|u_i\|_2^2 + \lambda \sum_{j=1}^n \|v_j\|_2^2 \tag{4}$$

where $\lambda > 0$ is the regularization coefficient. We will implement algorithms to learn vector representations by minimizing (4).

Since this is a non-convex optimization problem, the initial starting point and hyperparameters may affect the quality of $\widehat{R}$. You may need to tune λ and σ to optimize the loss you see.

- *Alternating minimization*: First, randomly initialize $\{u_i\}$ and $\{v_j\}$. Then, alternatate between (1) minimizing the loss function with respect to $\{u_i\}$ by treating $\{v_j\}$ as fixed; and (2) minimizing the loss function with respect to $\{v_j\}$ by treating $\{u_i\}$ as fixed. Repeat (1) and (2) until both $\{u_i\}$ and $\{v_j\}$ converge. Note that when one of $\{u_i\}$ or $\{v_j\}$ is given, minimizing the loss function with respect to the other part has a closed-form solution.
- Try $d \in \{1, 2, 5, 10, 20, 50\}$ and plot the mean squared error of train and test as a function of d.

Some hints: Common choices for initializing the vectors $\{u_i\}_{i=1}^m$, $\{v_j\}_{j=1}^n$ include: entries drawn from `np.random.rand()` scaled by some scale factor $\sigma > 0$ (σ is an additional hyperparameter), or using one of the solutions from part b or c. You should never be allocating an $m \times n$ matrix for this problem.

d. *[10 points]* Repeat part c, using batched SGD rather than alternating minimization.

Stochastic Gradient Descent: First, randomly initialize $\{u_i\}$ and $\{v_j\}$. Then take a batch of random samples (of size b from your training set and compute a gradient step, and repeat until convergence.

Note that batch size b, regularization constant λ, scaling parameter σ and learning rate η are all hyperparameters. Since this is a non-convex optimization, the results may be quite sensitive to these hyperparameters and to the initialization.

One strategy for choosing η is to select the largest constant value such that the loss L still tends to decrease. Another strategy is to pick a relatively large value of η and then scale it by some factor $\beta \in (0,1)$ so that $\eta \mapsto \beta\eta$ every time a number of examples are seen that exceeds the size of the training set.

Feel free to modify the loss function to, say, different regularizers if it helps reduce the test error. See `http://www.optimization-online.org/DB_FILE/2011/04/3012.pdf` for some ideas.

e. *[5 points]* Briefly, in words, compare the results of the prior two parts. This is an example where the loss functions are identical, but the *algorithm* used has drastic impact on how much the model fits, overfits, or generalizes to new, unseen data.

f. (Extra credit: *[5 points]*). Implement any algorithm you'd like (you must implement it yourself; do not use an off-the-shelf algorithm from e.g. `scikit-learn`) to find an estimator that achieves a test error of no more than 0.9. Please include your code.

A7. Consider the matrix $X^* \in \mathbb{R}^{n\times d}$ (assume $n \geq d$) with singular values $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_d$. Our goal in this problem is to find the best rank-k approximation of X^* via singular value decomposition, thus rigorously justifying the premise of the problem "Matrix Completion and Recommendation System". This goal can be formulated mathematically as the following optimization problem:

$$\min_{X:\text{rank}(X)=k} \|X - X^*\|_F^2 \tag{5}$$

a. We say a problem $\min_{x\in K} f(x)$ is convex if the search space K is a convex set and the objective function f is a convex function. Show that problem (5) is *not* a convex optimization problem. (Hint: is the search space convex?)

b. In this subproblem, we study projection matrices. Let $\{w_1, ..., w_k\} \in \mathbb{R}^n$ be a set of orthonormal vectors [3]. We can think of these as spanning the column space of a rank k matrix $X \in \mathbb{R}^{n\times d}$, like the ones considered in (5).

   (a) Let $X^*_{(i)}$ be the i'th column of X^*. Find the $v \in \text{span}\{w_1, \ldots, w_k\}$ that minimizes $\|X^*_{(i)} - v\|_2$.

   (b) Consider the matrix Π defined as $\Pi = \sum_{i=1}^k w_i w_i^\top$. What are the possible singular values of Π? Justify your answer with a proof.

   (c) Show that $\Pi^2 = \Pi$. Give a geometric interpretation of Trace(Π).

   (d) In addition to the vectors $\{w_i\}_{i=1}^k$ defined at the start of this sub-problem, consider the vectors $\{w_j\}_{j=k+1}^n$ obtained by "completing" the subspace $\mathbb{R}^n$. Show that there is a unique projection matrix $\Pi = \sum_{i=1}^n w_i w_i^\top$ of rank n, which is the identity matrix.

c. In this sub-problem, we use the projection matrix from the previous sub-problem to construct a rank k approximation.

   (a) Consider an orthonormal set of vectors $\{w_i\}_{i=1}^d \in \mathbb{R}^n$ that span the column space of X^*, with the first k vectors defined as in the previous sub-problem, and the remaining ones chosen to complete the subspace. As in the previous sub-problem, define $\Pi = \sum_{i=1}^k w_i w_i^\top$. Define $\widehat{X} = \Pi X^*$ and $\Pi^\perp = \sum_{i=k+1}^d w_i w_i^\top$. Show that

$$\|\widehat{X} - X^*\|_F^2 = \|\Pi^\perp X^*\|_F^2.$$

   (b) Show that $\|\Pi^\perp X^*\|_F^2 = \text{Trace}(X^* X^{*\top} \Pi^\perp)$.

   (c) Using that X^* can be written as $X^* = \sum_{i=1}^d \sigma_i u_i v_i^\top$, show that $\text{Trace}(X^* X^{*\top} \Pi^\perp) \geq \sum_{i=k+1}^d \sigma_i^2$.

d. In this sub-problem, we conclude our result for the Frobenius norm approximation.

[3] Recall that we say a set of vectors $u, v \in \mathbb{R}^n$ are orthonormal if $\|v\|_2 = \|u\|_2 = 1$ and $u^\top v = 0$.

(a) Show that $\min_{\text{rank}(X)=k} \|X - X^*\|_F^2 = \sum_{i=k+1}^{d} \sigma_i^2$.

(b) Based on the above sub-problems, describe a method to construct the best rank-k approximation of a matrix X^*, where we measure error in squared Frobenius norm.