ECE 661 - Homework 3

Michael Li (zl310)

October 19, 2023

I have adhered to the Duke Community Standard in completing this assignment.

Contents

List of Figures

1 True/False Questions (30 pts)

Problem 1.1 (3 pts) In the self-attention layer of Transformer models, we compute three core variables: key, value and query.

True. According to slide 11 lecture 10, attention is computed using queries, keys, and values. Specifically for self-attention, the queries, values, and keys are all different linear transformations of the same source.

Problem 1.2 (3 pts) In the self-attention layer of Transformer models, the attention is denoted by the cosine similarity between the key and query.

False. Attention is given by $a_{ij} = \frac{\exp(s_{ij})}{\sum_{j'} \exp(s_i j')}$ where $s_{ij} = q_i^T k_j$ for q_i, k_j being the query and the key. The term "cosine similarity" is a fancy way of saying "normalised dot product". Although attention uses the dot product of the query and the key (slide 16, lecture 10), the score is then passes through a softmax function, meaning that we cannot say that the attention is denoted by the cosine similarity between the key and the query. This paper `https://paperswithcode.com/method/scaled` implies that attention can also be named "scaled dot-product attention" (i.e., "scaled cosine similarity"). However, the softmax between the score and attention suggests that the attention is *not denoted by* the cosine similarity between key and value. Attention isn't proportional to the cosine similarity between key and value.

Problem 1.3 (3 pts) In the self-attention layer of Transformer models, after obtaining the attention matrix, we need to further apply a normalization on it (e.g., layer normalization or batch normalization).

False. According to slide 8 lecture 10, softmax is applied to the attention score to get the attention distribution which is the attention matrix. Since we've just applied "normalisation" (i.e., softmax) on the attention scores to the attention matrix, it does not make sense to apply layer/batch normalisation directly on the attention matrix. The normalisation modules in the figure on slide 20, lecture 10 most likely normalise the output of the multi-head attention block and not the attention matrix itself. Slide 12 lecture 10 indicates that there's a softmax operation required to get from the score to the attention, which means that normalisation is already applied to calculate attention.

Problem 1.4 (3 pts) The encoder of Transformer learns auto-regressively.

False. The encoder of a transformer learns through autoencoding. According to slide 26 lecture 10, BERT is autoencoding. Since BERT is an example of a pretrained-encoder and encoders can condition on future words, encoders do not learn auto-regressively.
Furthermore, consider the definition of learning auto-regressively: it means that only timestamps in the past are considered when predicting the output at the next timestamp. The encoder of a transformer certainly considers future timestamps when undergoing training, meaning that it cannot be auto-regressive.

Problem 1.5 (3 pts) Both BERT's and GPT's architecture are Transformer decoder.

False. According to slide 26 lecture 10, BERT is an example of a pre-trained encoder, while GPT2 is an example of a pre-trained decoder. So, BERT is just an encoder and GPT models only use decoders.

Problem 1.6 (3 pts) BERT's pre-training objectives include (a) masked token prediction (masked language modeling) and (b) next sentence prediction.

True. According to slide 28 lecture 10, the two pre-training objectives of BERT are: (a) masked language model which aims to predict masked words, and (b) next sentence prediction which aims to predict whether two sentences are adjacent to each other in the text.

Problem 1.7 (3 pts) Both GPT and BERT are zero-shot learner (can be transferred to unseen domain and tasks).

True. Both GPT and BERT can generalise to unseen data after fine-tuning. However, some amount of fine-tuning is required for GPT/BERT to be transferred to unseen tasks and BERT suffers from the problem of pre-training-finetune discrepancy. The important thing is that both BERT and GPT *can* be zero-shot learners. It is of course always better to perform some fine-tuning before applying either GPT or BERT to the specific tasks ("few-shot"). A withdrawn paper by Wang et al. (`https://openreview.net/forum?id=YLglAn-USkf`) shows that useful techniques (for example, inserting masks) can be applied such that BERT performs well in zero-shot applications. This OpenAI paper linked in lecture 10 `https://arxiv.org/pdf/2005.14165.pdf` suggests that GPT models *can be* zero-shot learners in the sense that it performs quite well in zero-shot situations. This of course does not mean that the results are stellar when the model is applied in a zero-shot situation. Therefore, it's correct to say that they can be both "zero-shot" and "few-shot" learners depending on the application.

Problem 1.8 (3 pts) Gradient clipping can be used to alleviate gradient vanishing problem.

False. Gradient clipping sets a range within which the gradient can vary. This means that the absolute value of the gradient cannot exceed a certain upper limit. Normally, gradient clipping is used solve the problem of exploding gradients according to slide 50, lecture 5. Since PyTorch does not implement a floor for the norm with their function `nn.utils.clip_grad_norm_`, it is **normally not the case that gradient clipping can also solve the problem of vanishing gradients**. Methods for alleviating the gradient vanishing problem normally include using ReLU as the activation layer, using the LSTM module, or using attention (depending on the context) according to slide 31 lecture 9 and slide 9 lecture 10.

Problem 1.9 (3 pts) Word embeddings can contain positive or negative values.

True. Slide 16 lecture 9 demonstrates valid word embeddings for the words "fox" and "jump". There is no restriction on the sign of each coordinate in an embedding. The PyTorch documentation on `nn.Embedding` also does not place restrictions on the sign of coordinates within each embedding vector (according to `https://pytorch.org/docs/stable/generated/torch.nn.Embedding.html`).

Problem 1.10 (3 pts) The memory cell of an LSTM is computed by a weighted average of previous memory state and current memory state where the sum of weights is 1.

False. The memory cell is given by $\mathbf{C}_t = \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \tilde{\mathbf{C}}_t$ where $\mathbf{I}_t$ determines how much we take new data $\tilde{\mathbf{C}}_t$ into account, and the forget gate $\mathbf{F}_t$ determines how much is retained from the old cell's internal state $\mathbf{C}_{t-1}$ (according to `https://d2l.ai/chapter_recurrent-modern/lstm.html`). Here, $\mathbf{I}_t, \mathbf{F}_t$ can be considered as the weights.
Since the input and forget gates ($\mathbf{I}_t, \mathbf{F}_t$ respectively) are the output of a sigmoid (not of a softmax), each individual value within the matrices $\mathbf{I}_t, \mathbf{F}_t$ are indeed between 0 and 1. However, this does not mean that the sum of the weights (i.e., the sum of the elements in $\mathbf{I}_t, \mathbf{F}_t$)

2 Lab 1: Recurrent Neural Network for Sentiment Analysis (45 pts)

(a) (5 pts) Implement your own data loader function. First, read the data from the dataset file on the local disk. Then split the dataset into three sets: train, validation, and test by `7:1:2` ratio. Finally return `x_train, x_valid, x_test, y_train, y_valid` and `y_test`, where `x` represents reviews and `y` represent labels.

This is implemented in the notebook. It is copied here for your convenience. The entire data set is read and split using the `train_test_split` package from `scikit-learn`. First, I split the data into the train and (test-validation) sets. Then, I split the (test-validation) set into the test and validation sets. This function is only called once throughout all iterations of training and the random state is the same throughout the entire lab.

```
def load_imdb(base_csv:str = './IMDBDataset.csv'):
    """
    Load the IMDB dataset
    :param base_csv: the path of the dataset file.
    :return: train, validation and test set.
    """
    # Add your code here.
    df = pd.read_csv('IMDBDataset.csv')
    x = list(df['review'])
    y = list(df['sentiment'])
    # 7:3 train/(test&Val) ratio
    x_train, x_testVal, y_train, y_testVal = train_test_split(x, y, test_size=0.3,
                                                              random_state=1002,
                                                              shuffle=True)
    # 1:2 val='train'/test ratio
    x_valid, x_test, y_valid, y_test = train_test_split(x_testVal, y_testVal,
                                                        test_size=float(2/3),
                                                        random_state=1002,
                                                        shuffle=True)
    # I'm only to run this function once, so it shouldn't matter across different trainings if I shuffle.
    # random seed is set, so should be constant across different kernel sessions.
    print(f'shape of train data is {len(x_train)}')
    print(f'shape of test data is {len(x_test)}')
    print(f'shape of valid data is {len(x_valid)}')
    return x_train, x_valid, x_test, y_train, y_valid, y_test
```

(b) (5 pts) Implement the `build_vocab` function to build a vocabulary based on the training corpus. You should first compute the frequency of all the words in the training corpus. Remove the words that are in the `STOP_WORDS`. Then filter the words by their frequency ($\geq$ `min_freq`) and finally generate a corpus variable that contains a list of words.

This is implemented in the notebook. It is copied here for your convenience. The tokenization process is very similar to the one in the `tokenize` function. Each input is split using the `.split()` method, converted to lowercase, and appended to the list of words. The word-frequency is counted using a `Counter`.

```
def build_vocab(x_train:list, min_freq: int=5, hparams=None) -> dict:
    """
    build a vocabulary based on the training corpus.
    :param x_train:  List. The training corpus. Each sample in the list is a string of text.
    :param min_freq: Int. The frequency threshold for selecting words.
    :return: dictionary {word:index}
    """
    # Add your code here. Your code should assign corpus with a list of words.

    # split x_train into words
    words_temp = []
    for sent in x_train:
        words_lst = sent.split()
        words_lst_lower = [w.lower() for w in words_lst]
        words_temp.extend(words_lst_lower)
    # no additional string preprocessing

    corpus = Counter([w for w in words_temp if w not in hparams.STOP_WORDS])

    # sorting on the basis of most common words
    # corpus_ = sorted(corpus, key=corpus.get, reverse=True)[:1000]
    corpus_ = [word for word, freq in corpus.items() if freq >= min_freq]
    # creating a dict
    vocab = {word : idx + 2 for idx, word in enumerate(corpus_)}
    vocab[hparams.PAD_TOKEN] = hparams.PAD_INDEX
    vocab[hparams.UNK_TOKEN] = hparams.UNK_INDEX
    return vocab
```

(c) (5 pts) Implement the `tokenization` function. For each word, find its index in the vocabulary. Return a list of integers that represents the indices of words in the example.
This is implemented in the notebook. It is copied here for your convenience. Note that I do not substitute unknown words for `hparams.UNK_TOKEN` in my `tokenize` function. This means that if a lowercase word in `example.split()` does not exist in `vocab`, it is simply skipped over. This avoids crowding out certain validation/test inputs with `hparams.UNK_TOKEN` (i.e., avoids inputs such as $\{1, 1, 1, 1, 2, 91, 1, 1, 1\}$). The model never sees `hparams.UNK_TOKEN` during training and removing unknown words isn't introducing data leakage into the model since the process of removing words doesn't introduce new information into the validation/testing data (information is removed from the validation/test sets).

```
def tokenize(vocab: dict, example: str) -> list:
    """
    Tokenize the give example string into a list of token indices.
    :param vocab: dict, the vocabulary.
    :param example: a string of text.
    :return: a list of token indices.
    """
    # Your code here.
    ex_tok = example.split()
    ex = [w.lower() for w in ex_tok]
    return [vocab[w] for w in ex if w in vocab]
```

(d) (5 pts) Implement the `__getitem__` function in the IMDB class. Given an index i, you should return the i-th review and label. The review is originally a string. Please tokenize it into a sequence of token indices. Use the `max_length` parameter to truncate the sequence so that it contains at most `max_length` tokens. Convert the label string ('positive'/'negative') to a binary index, such as 'positive' is 1 and 'negative' is 0. Return a dictionary containing three keys: `ids`, `length`, `label` which represent the list of token ids, the length of the sequence, the binary label.
This is implemented in the notebook. It is copied here for your convenience. Each review is brought out of `vocab` and tokenized, which is then truncated conditioned on the length.

```
def __getitem__(self, idx: int) -> Dict:
    pn_dict = {'positive': 1, 'negative': 0}

    review, binary_label = tokenize(vocab=self.vocab, example=self.x[idx]), pn_dict[self.y[idx]]
    # print(len(review), review[0:2])
    if len(review) > self.max_length:
        return {'ids': review[0:self.max_length], 'length': self.max_length, 'label': binary_label}
    else:
        return {'ids': review, 'length': len(review), 'label': binary_label}
```

(e) (10 pts) Implement the LSTM model for sentiment analysis.
(e.1) (5 pts) In `__init__`, an LSTM model contains an embedding layer, an `lstm cell`, a linear layer, and a dropout layer. You can call functions from Pytorch's nn library. For example, `nn.Embedding`, `nn.LSTM`, `nn.Linear`.
(e.2) (5 pts) In forward, decide where to apply dropout. The sequences in the batch have different lengths. Write/call a function to pad the sequences into the same length. Apply a fully-connected (fc) layer to the output of the LSTM layer. Return the output features which is of size (`batch size`, `output dim`).

This is implemented in the notebook. It is copied here for your convenience. If the `nn.LSTM` module only has 1 layer, there will be no dropout and the LSTM module seems to complain when dropout $\neq 0.0$. A separate layer is added for dropout in the last layer and only when `n_layers > 0` is the `dropout_rate` parameter passed into the LSTM module. In addition, if `bidirectional == True`, the number of input features in the fully connected layer increases by a factor of 2.
In the `forward` method, the input is padded and the embeddings are generated. The input (i.e., emeddings) to the LSTM layer is packed; the output of the LSTM layer is unpacked, resulting in a padded tensor of shape (`batch_size`, `max_seq_len`, `hideen_dim`). Since the output of the `nn.utils.rnn.pad_packed_sequence` is padded, the indexing in the dropout layer `out[range(out.shape[0]), length.int() - 1, :]` needs to take the last non-padded hidden dimension for each output in the batch, resulting in a tensor of shape (`batch_size`, `hidden_dim`). The dropout layer is applied on the output of the LSTM layer. The dropout layer's output is fed into the fully-connected layer.

```
class LSTM(nn.Module):
    def __init__(self, vocab_size: int, embedding_dim: int, hidden_dim: int,
        output_dim: int, n_layers: int, dropout_rate: float, pad_index: int,
        bidirectional: bool, **kwargs):
        super().__init__()
        self.pad_index = pad_index
        self.hidden_dim = hidden_dim
        self.embedding = nn.Embedding(num_embeddings=vocab_size,
                                      embedding_dim=embedding_dim, padding_idx=pad_index,
                                      max_norm=None, norm_type=2.0, scale_grad_by_freq=False,
                                      sparse=False, _weight=None, _freeze=False,
                                      device=None, dtype=None)
        lstm_dropout = None
        if n_layers > 1:
            lstm_dropout = dropout_rate
        else:
            lstm_dropout = 0.0
        self.lstm = nn.LSTM(input_size=embedding_dim, hidden_size=hidden_dim,
                            num_layers=n_layers, bias=True, batch_first=True,
                            dropout=lstm_dropout, bidirectional=bidirectional, proj_size=0,
                            device=None, dtype=None)

        self.dropout = nn.Dropout(dropout_rate)

        self.fc = nn.Linear(in_features=hidden_dim, out_features=output_dim,
                            bias=True, device=None, dtype=None)
        if bidirectional: # bidirectional: 2*hidden_dim is needed
            # since lstm output is double the size
            self.fc = nn.Linear(in_features=2*hidden_dim, out_features=output_dim,
                                bias=True, device=None, dtype=None)
        # Weight initialization. DO NOT CHANGE!
        if "weight_init_fn" not in kwargs:
            self.apply(init_weights)
        else:
            self.apply(kwargs["weight_init_fn"])

    def forward(self, ids:torch.Tensor, length:torch.Tensor):
        # note: using batch_first=True
        # do we need to pad? the dataloader seems to be pretty good at padding
        # NO NEED to use nn.utils.rnn.pad_sequence since collate_fn already pads everything
        # and each batch has already been padded.
        padded_ids = nn.utils.rnn.pad_sequence(sequences=ids, batch_first=True,
                                               padding_value=self.pad_index)

        # Add your code here.
        out = self.embedding(padded_ids)
        # packs the embeddings (better for computation)
        out = nn.utils.rnn.pack_padded_sequence(out, length, batch_first=True, enforce_sorted=False)

        # Pass embedded input through the LSTM and dropout layers
        out, hidden = self.lstm(out)
        # otuput of lstm is packed sequence
        out, _ = nn.utils.rnn.pad_packed_sequence(out, batch_first=True,
                                                  padding_value=self.pad_index)
        # unpack the packed thing, should result in tensor of shape (batch size, seq len, hidden_size)
        # link to explain the range indexing below: take out every length-1-th element for each batch
        # https://stackoverflow.com/questions/53123009/using-python-range-objects-to-index-into-numpy-arrays
        out = self.dropout(out[range(out.shape[0]), length.int() - 1, :])
        # dropout(tensor of shape [batch_size, hidden_dim]) i.e. apply dropout on
        # last hidden state of each input in the batch

        prediction = self.fc(out)
        # Get the output from the last time step
        return prediction
```

(f) (5 pts) As a sanity check, train the LSTM model for 5 epochs with the `SGD` **optimiser**. **Do you observe a steady and consistent decrease of the loss value as training progresses?**
As inferred from Figure 1, there exists no steady and consistent decrease of the loss value as training progresses. Note: if you would like to see the lack of change in loss, please refer to the Jupyter Notebook.
Please see the training process in the Jupyter Notebook attached below.

Report your observation on the learning dynamics of training loss, and validation loss on the `IMDB` dataset. Do they meet your expectations and why?
According to Figure 1 which reports the training and validation accuracy (from which the loss can be inferred), the LSTM model fails to train. The loss does not decrease and stays at around ln(2).
This meets expectations because we aren't using adaptive optimisers when training the LSTM model. It is still difficult for an SGD optimiser with momentum to navigate out of a local minimum or a saddle point. Since the SGD optimiser has difficulties navigating the very complicated loss surface, the descent is very slow, an issue which gradient vanishing contributes to. According to the two gifs in this link (`https://www.ruder.io/optimizing-gradient-descent/`), the SGD optimiser is very slow with descending to the global minimum and does not leave a saddle point. This means that the model equipped with the SGD optimiser would take a very large number of epochs before it trains when no saddle points exist or it wouldn't train at all if the model ends up on a saddle point at some point (Dauphin, Y., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S., & Bengio, Y. (2014). Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. arXiv, 1–14. Retrieved from `http://arxiv.org/abs/1406.2572`).

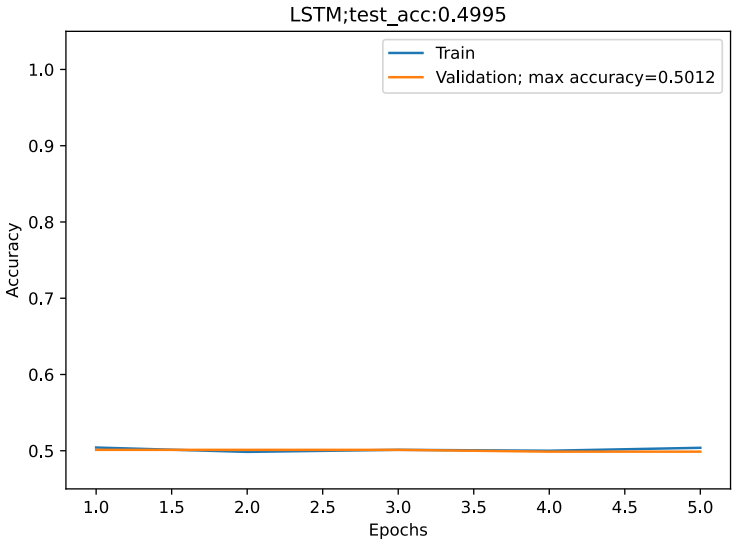


Figure 1: Training an LSTM using default hyperparameters.

(g) (10 pts) Implement the GRU model.
(g.1) (5 pts) In `__init__`, a GRU model includes an embedding layer, an gated recurrent unit, a linear layer, and a dropout layer. You can call functions from `torch.nn` library. For example, `nn.Embedding`, `nn.GRU`, `nn.Linear`.
(g.2) (5 pts) In forward, decide where to apply dropout. The sequences in the batch have different lengths. Write/call a function to pad the sequences into the same length. Apply a Fully-Connected (FC) layer to the output of the GRU layer. Return the output features which is of size (`batch size`, `output dim`).

This is implemented in the notebook. It is copied here for your convenience. The logic of the code is very similar to the LSTM code so I do not repeat the explanation here.

```
class GRU(nn.Module):
    def __init__(self, vocab_size: int, embedding_dim: int, hidden_dim: int, output_dim: int,
    n_layers: int, dropout_rate: float, pad_index: int, bidirectional: bool, **kwargs):
        super().__init__()
        # Add your code here. Initializing each layer by the given arguments.
        self.pad_index = pad_index
        self.embedding = nn.Embedding(num_embeddings=vocab_size,
                                      embedding_dim=embedding_dim, padding_idx=pad_index,
                                      max_norm=None, norm_type=2.0, scale_grad_by_freq=False,
                                      sparse=False, _weight=None, _freeze=False,
                                      device=None, dtype=None)
        gru_dropout = None
        if n_layers > 1:
            gru_dropout = dropout_rate
        else:
            gru_dropout = 0.0

        self.gru = nn.GRU(input_size=embedding_dim, hidden_size=hidden_dim,
                          num_layers=n_layers, bias=True, batch_first=True,
                          dropout=gru_dropout, bidirectional=bidirectional,
                          device=None, dtype=None)
        self.fc = nn.Linear(in_features=hidden_dim, out_features=output_dim,
                            bias=True, device=None, dtype=None)
        if bidirectional:  # hidden_dim*2, same reason as in LSTM
            self.fc = nn.Linear(in_features=hidden_dim*2, out_features=output_dim,
                                bias=True, device=None, dtype=None)
        self.dropout = nn.Dropout(dropout_rate)
        # Weight Initialization. DO NOT CHANGE!
        if "weight_init_fn" not in kwargs:
            self.apply(init_weights)
        else:
            self.apply(kwargs["weight_init_fn"])

    def forward(self, ids:torch.Tensor, length:torch.Tensor):
        # Add your code here.
        # DO NOT need to use nn.utils.rnn.pad_sequence since collate_fn already pads everything
        # and each batch has already been padded.
        padded_ids = nn.utils.rnn.pad_sequence(sequences=ids, batch_first=True,
                                               padding_value=self.pad_index)
        out = self.embedding(padded_ids)
        out = nn.utils.rnn.pack_padded_sequence(out, length, batch_first=True, enforce_sorted=False)

        out, _ = self.gru(out)  # second output is hidden state

        # unpack gru_out since it's packed, results in tensor of shape (batch size, seq len, hidden_size)
        out, _ = nn.utils.rnn.pad_packed_sequence(out, batch_first=True,
                                                  padding_value=self.pad_index)
        # Apply dropout to the GRU output
        out = self.dropout(out[range(out.shape[0]), length.int() - 1, :])
        # array indexing:
        # for each input, take the last non-padded hidden state (which is where length comes in).
        # ends up with tensor of shape [batch size, hidden_dim]
        prediction = self.fc(out)
        return prediction
```

(h) (Bonus, 5 pts) Repeat (f) for GRU model and report your observations on the training curve. What do you hypothesize is the issue that results in the current training curve?

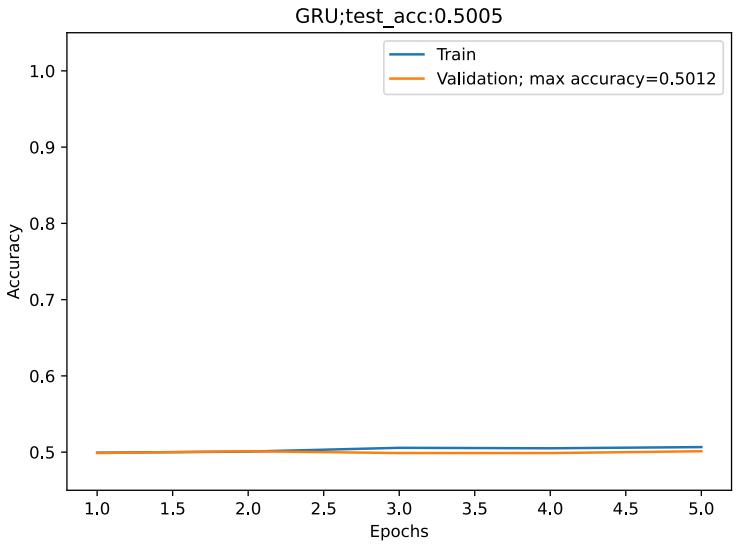
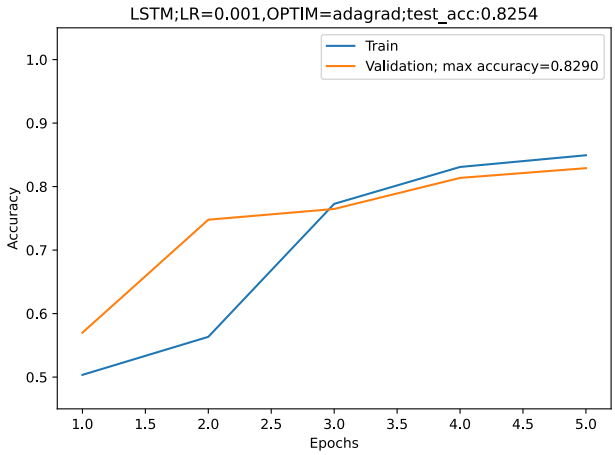


Figure 2: Training using a GRU using default hyperparameters.

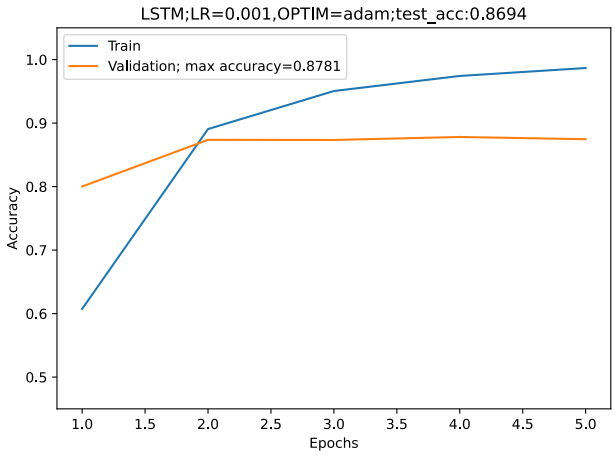
According to Figure 2, the training curve is flat, which means that the model isn't training at all in five epochs. The issue is that we're using an unsuitable optimiser, namely the SGD optimiser, which gets stuck on saddle points and local minima due to gradient vanishing. The recommended optimiser (according to slide 31, lecture 9) is RMSprop for training RNNs, including LSTMs and GRUs. Therefore, if we change the optimiser, we will see that the model trains.

3 Lab 2: Training and Improving Recurrent Neural Network (25 pts)

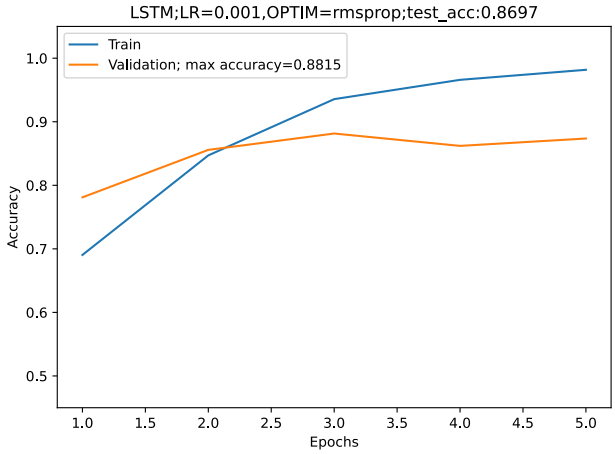
(a) (5 pts) We start to look at the optimisers in training RNN models. As SGD might not be good enough, we switch to advanced optimisers (i.e., Adagrad, RMSprop, and Adam) with adaptive learning rate schedule. We start with exploring these optimisers on training an LSTM. You are asked to employ each of the optimiser on LSTM and report your observations. **Which optimiser gives the best training results, and why?**



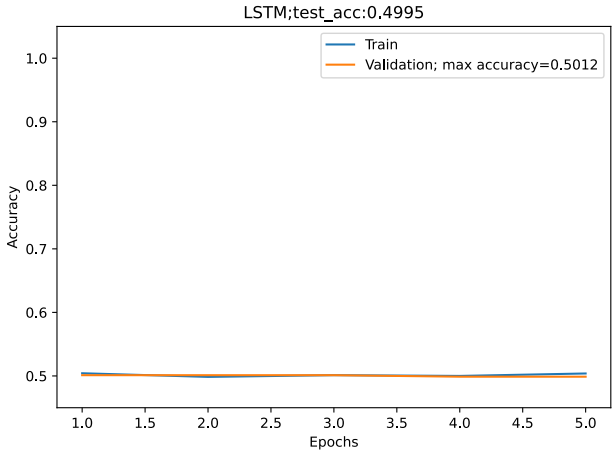
(a) Training an LSTM using Adagrad, $lr = 0.001$.



(b) Training an LSTM using Adam, $lr = 0.001$.



(c) Training an LSTM using RMSprop, $lr = 0.001$.

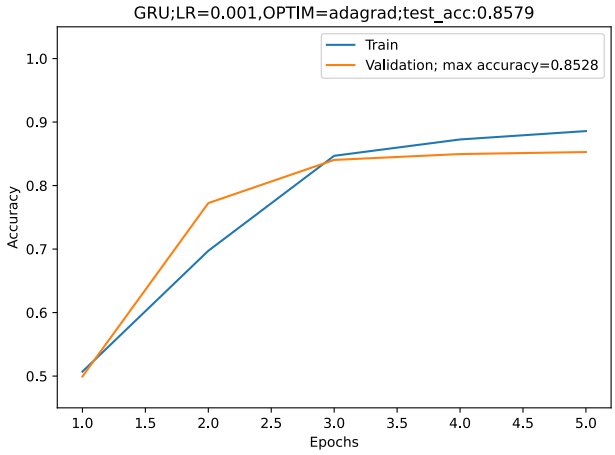


(d) Training an LSTM using default hyperparameters.

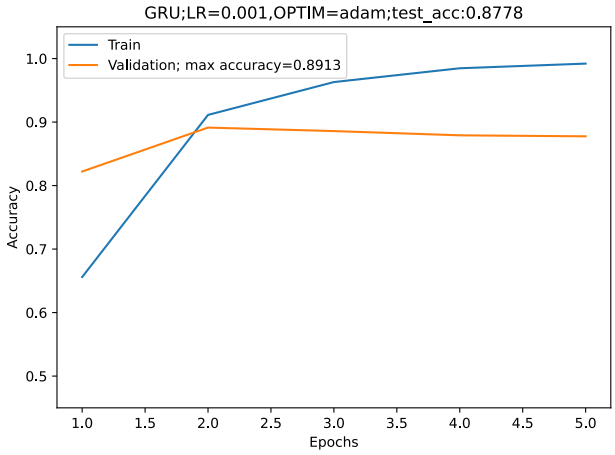
Figure 3: Variation of training/validation accuracy vs epochs with different optimisers (LSTM).

The training curves look much better when using adaptive optimisers (Figures 3a, 3b, 3c) compared to using the SGD optimiser (Figure 3d). The model actually trains and the training/validation accuracy values increase steadily. Adagrad performs poorly because the accumulation of gradients in the denominator of $W^{t+1} = W^t - \frac{\alpha \nabla_W l(W^t)}{\sqrt{G+\epsilon}}, G = \sum_{i=0}^{t} (\nabla_W l(W^t))^2$ grows too quickly, which shrinks the learning rate too quickly and leads to slow convergence (slide 46, lecture 5). The **RMSprop optimiser gives the best results** (Figure 3c, highest validation and test accuracy values) because it prevents learning rate shrinkage by using a moving average of incoming gradients in the denominator (slide 47, lecture 5). This prevents learning rate shrinkage and leads to faster convergence. The Adam optimiser, which draws from both SGD and RMSprop, also performs very well as it uses a lot of statistics. However, Adam as well as RMSprop oscillate at the end of convergence (slide 48, lecture 5).

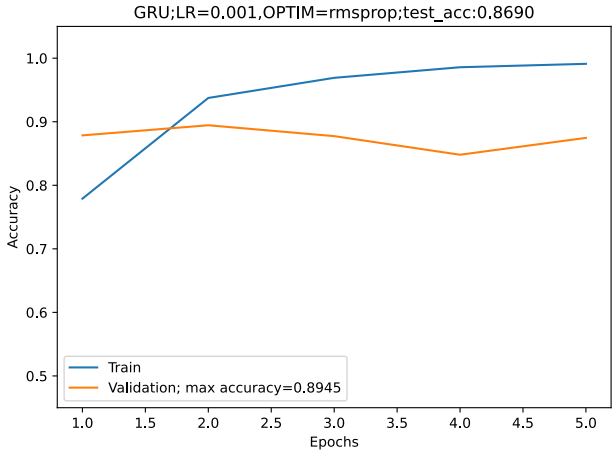
(b) (5pts) Repeat (a) on training your GRU model and provide an analysis on the performance of different optimisers on a GRU model. How does a GRU compare to an LSTM in terms of model performance and efficiency?



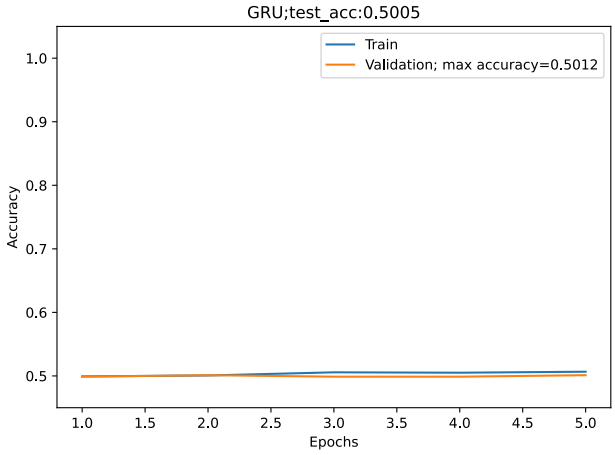
(a) Training a GRU using Adagrad, $lr = 0.001$.



(b) Training a GRU using Adam, $lr = 0.001$.



(c) Training a GRU using RMSprop, $lr = 0.001$.



(d) Training a GRU using default hyperparameters.

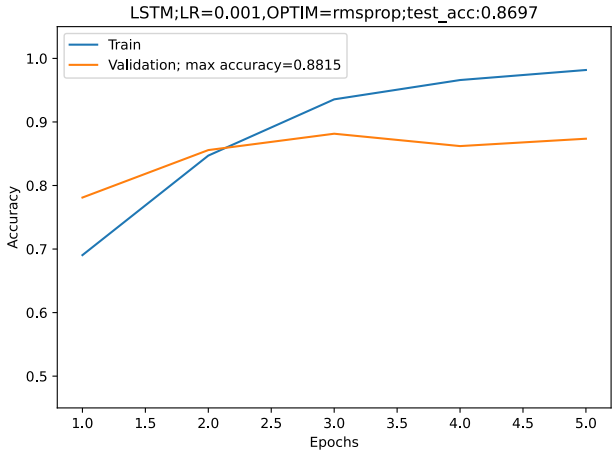
Figure 4: Variation of training/validation accuracy vs epochs with different optimisers (GRU).

We can see that the GRU using Adagrad (Figure 4a) performs worse that the two GRUs with Adam and RMSprop (Figures 4b, 4c). All adaptive optimisers perform better than vanilla GRU (Figure 4d.

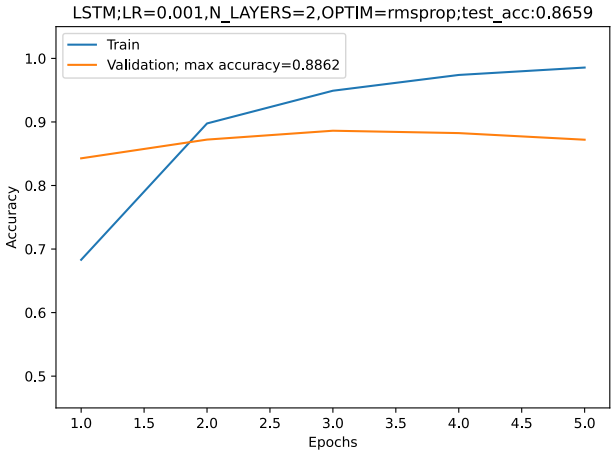
Performance: the performance of the GRU using Adagrad is much better than that of the LSTM using Adagrad and we see that the GRU (Adagrad) trains faster than the LSTM (Adagrad) (refer to Figures 3a, 4a). Although the accuracy on the testing data for both LSTMs and GRUs that use RMSprop/Adam are very similar, the highest validation accuracy values of the two GRU models (RMSprop/Adam) are higher than those of the two LSTM models (RMSprop/Adam). This could mean that the GRU models train faster than the LSTM models when given the same hyperparameters and optimisers. The GRUs using RMSprop/Adam over-fit towards the end, which could explain why validation accuracy is higher than test accuracy.

Efficiency: LSTM models have more trainable parameters (98131 trainable parameters for all three adaptive optimisers) than GRU models (87831). This suggests that the GRU models are slightly more efficient than the LSTM models since the smaller number of trainable parameters suggest that GRUs have demand less computational and memory resources.

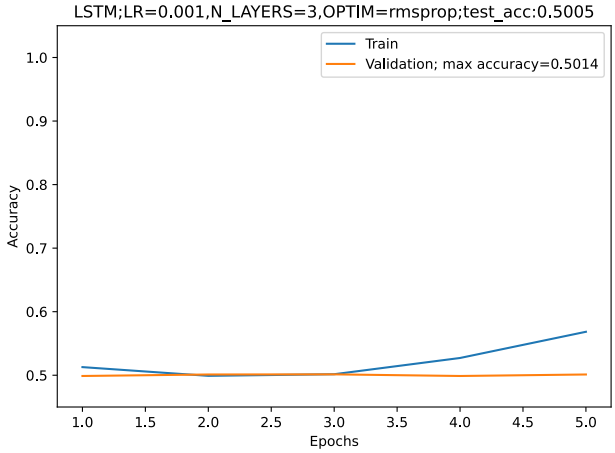
(c) (5 pts) Try to make your RNN model deeper by changing the number of layers. Is your RNN model achieving a better accuracy on IDMB classification? You may use LSTM as an example.



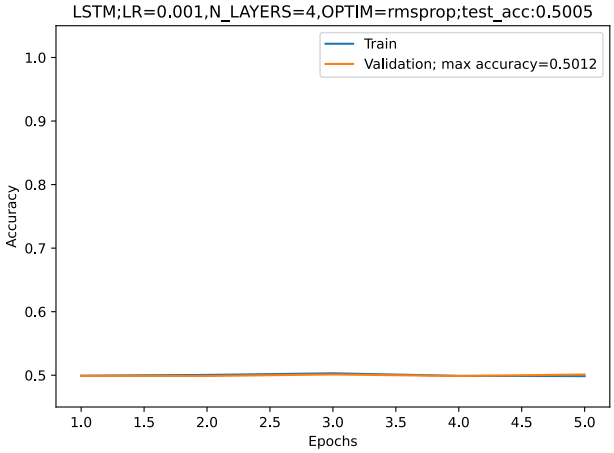
(a) Training an LSTM (RMSprop), `N_LAYERS=1`, 98131 parameters.



(b) Training an LSTM (RMSprop), `N_LAYERS=2`, 178931 parameters.



(c) Training an LSTM (RMSprop), `N_LAYERS=3`, 259731 parameters.

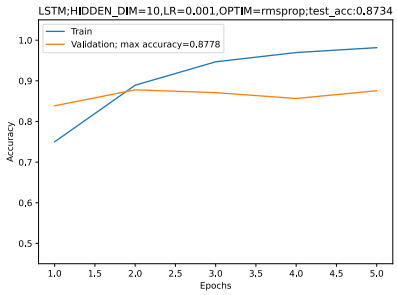


(d) Training an LSTM (RMSprop), `N_LAYERS=4`, 340531 parameters.

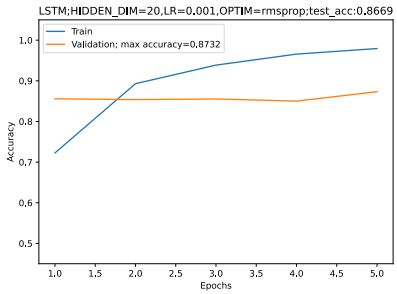
Figure 5: Variation of training/validation accuracy vs epochs with RMSprop, $lr = 0.001$, and different `N_LAYERS` (LSTM).

The LSTM model using `N_LAYERS=2` (Figure 5b) achieves similar accuracy values as `N_LAYERS=1` in Figure 5a (which is the default) . However, the model barely trains when `N_LAYERS=3` (Figure 5c) and does not train at all when `N_LAYERS=4` (Figure 5d). One explanation is that one part of the model is very complex, which is represented by the large number of layers, but other parts of the model are relatively simple (e.g., `EMBEDDING_DIM=1`, `HIDDEN_DIM=100` which is quite low). Although deep models are not themselves overly complex, the imbalance in complexity within the model may cause the model to not train well. This is very dependent on the random state within `load_imdb` when splitting the data. Choosing a different random state caused my `N_LAYERS=3,4` to be much better than what I've shown here in Figure 5.
Please refer to Figure 8a to visualise the trend in testing accuracy. Please refer to the Jupyter Notebook below for the code.

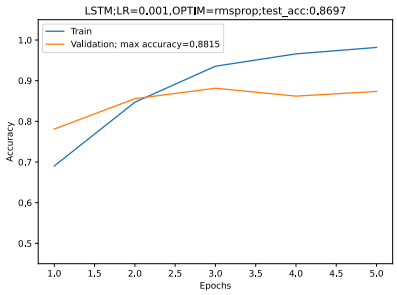
(d) (5 pts) Try to make your RNN model wider by changing the number of hidden units. Is your RNN model achieving a better accuracy on IDMB classification? You may use LSTM as an example.



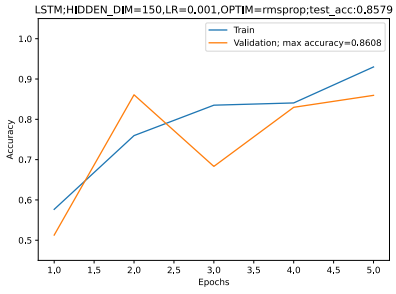
(a) 57271 params



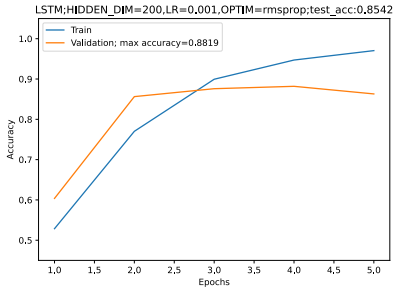
(b) 58611 params



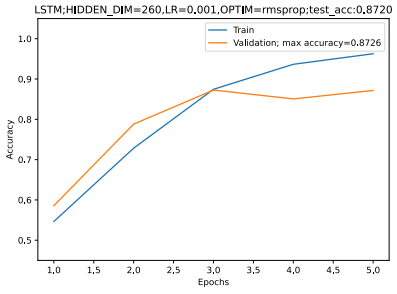
(c) Default, 98131 params.



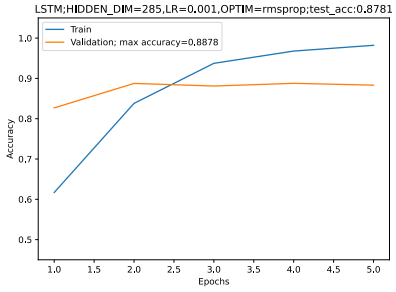
(d) 148831 params



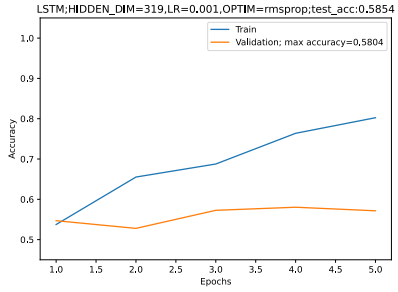
(e) 219531 params



(f) 330771 params



(g) 385621 params

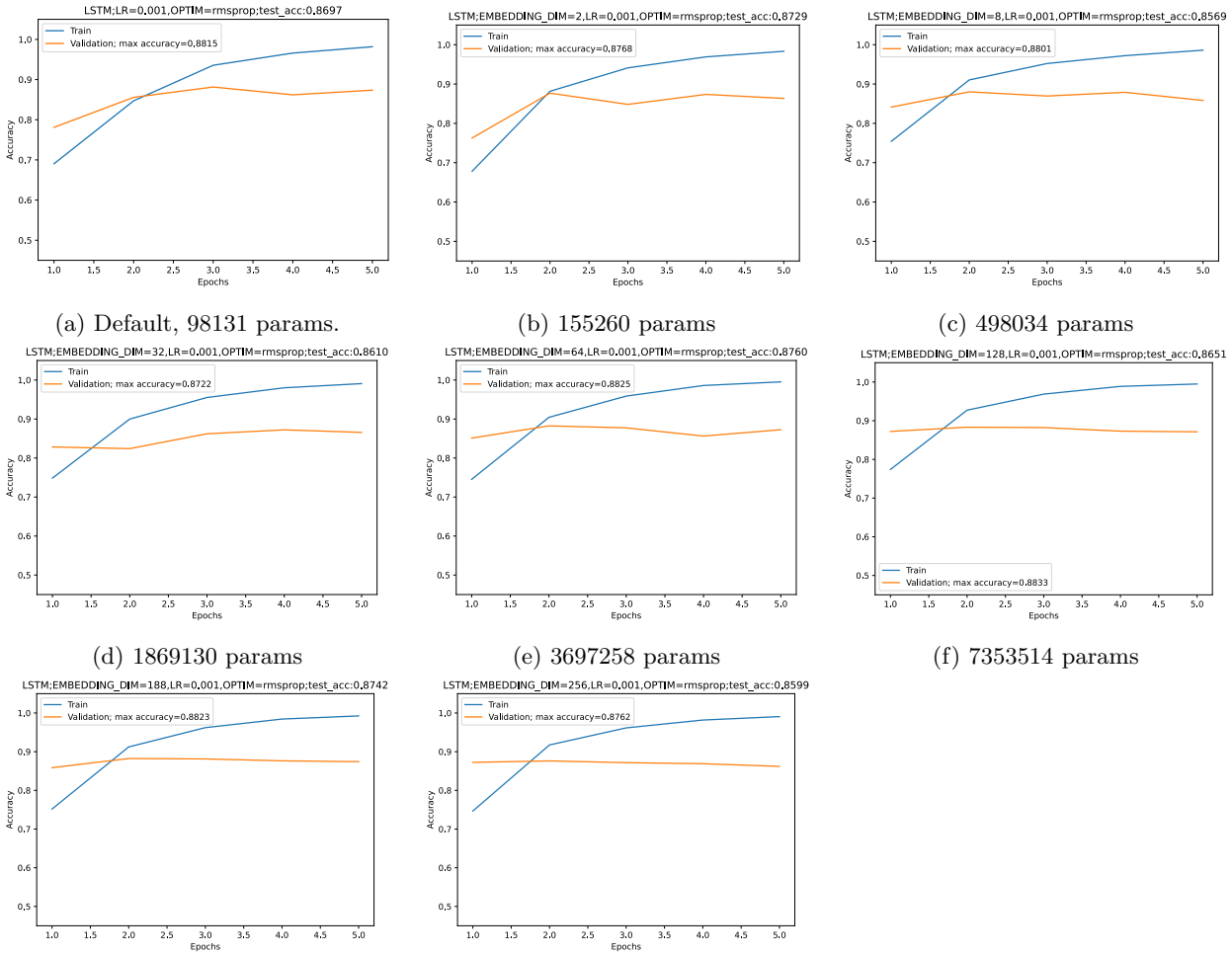


(h) 468241 params

Figure 6: Variation of training/validation accuracy vs epochs with RMSprop, $lr = 0.001$, and different `HIDDEN_DIM` (LSTM).

The model accuracy with `HIDDEN_DIM < 100` (Figures 6a, 6b) are comparable to the default model with `HIDDEN_DIM = 100` (Figure 6c). If I interpolate the accuracy data, models with intermediate `HIDDEN_DIM` values (roughly 150-250, Figures 6d and 6e) have slightly lower test accuracy than the default model. The model accuracy with `HIDDEN_DIM = 285` peaks at 0.8781 test accuracy. The very wide model with `HIDDEN_DIM=319` (Figure 6h) struggles to learn and has the lowest train/validation/test accuracy values. One also observes that models with higher `HIDDEN_DIM` values take longer to train since more complex models would naturally require more epochs to train the many parameters.
Please refer to Figure 8b to visualise the trend in testing accuracy. Please refer to the Jupyter Notebook below for the code.

(e) (5 pts) Embedding tables contain rich information of the input words and help build a more powerful representation with word vectors. Try to increase the dimension of embeddings. Is your RNN model achieving a better accuracy on IMDB classification? You may use LSTM as an example.



(a) Default, 98131 params.

(b) 155260 params

(c) 498034 params

(d) 1869130 params

(e) 3697258 params

(f) 7353514 params

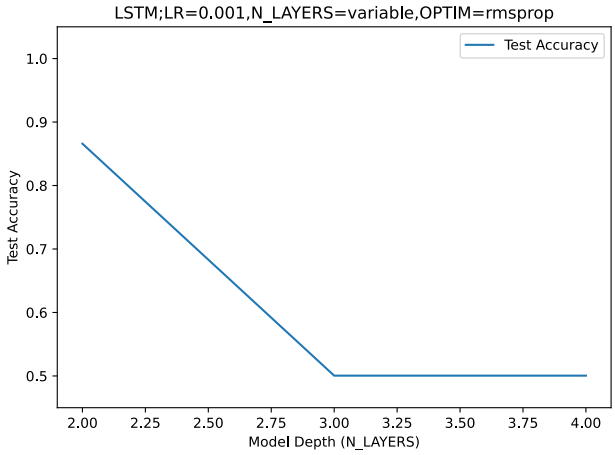
(g) 10781254 params

(h) 14666026 params

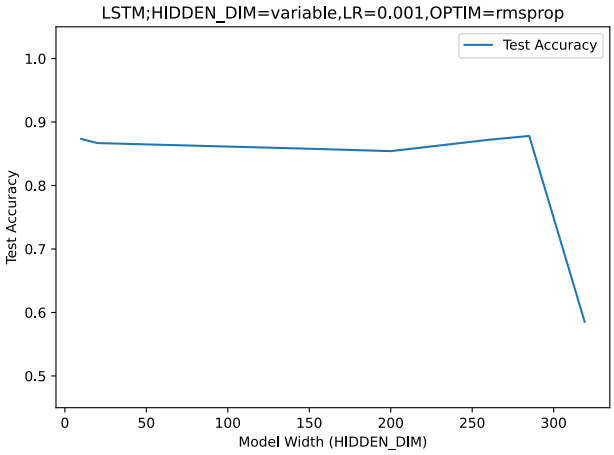
Figure 7: Variation of training/validation accuracy vs epochs with RMSprop, $lr = 0.001$, and different `EMBEDDING_DIM` (LSTM).

The RNNs in Figure 7 all perform very similarly. Only changing `EMBEDDING_DIM` doesn't seem to have much of an effect on the performance of the model. The training speed does increase. However, the number of parameters of each model increases as the value of `EMBEDDING_DIM` goes up. This means that, for models with higher `EMBEDDING_DIM`, more resources are consumed during training without a similar increase in performance.
Please refer to Figure 8c to visualise the trend in testing accuracy. Please refer to the Jupyter Notebook below for the code.

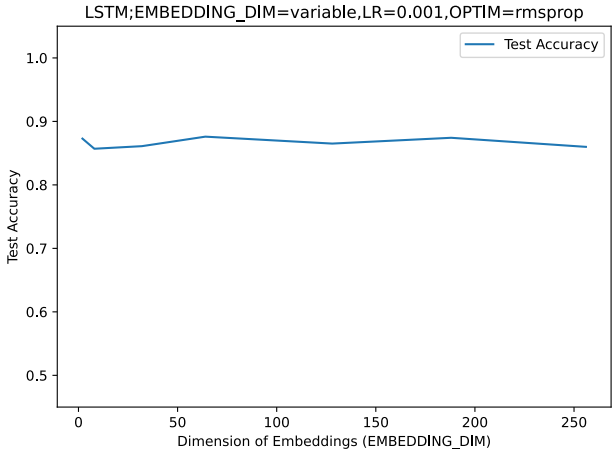
Figure 8 visualises the model performance for parts (c) to (e).
I chose the RMSprop optimiser because it gave the best performance when training LSTMs in lab 2 part (a). I used $lr = 0.001$ since the RMSprop optimiser is an adaptive optimiser.



(a) Training LSTMs with different model depth (`N_LAYERS`, part (c)).



(b) Training LSTMs with different model width (`HIDDEN_DIM`, part (d)).



(c) Training LSTMs with different dimensions for embeddings (`EMBEDDING_DIM`, part (e)).

Figure 8: Overall performance for lab 2 parts (c) - (e).

(f) (Bonus, 5 pts) A better way to scale up RNN models is simultaneously scaling up the number of hidden units, number of layers, and embedding dimension. This is called compound scaling, which is widely adopted in emerging ML research. You are asked to make no more than 50 trials to perform a compound scaling on your implemented RNN models. Is the model crafted via compound scaling performing better than the models you obtain in (d), (e), and (f)? You may use LSTM as an example.

Using the **ratio** of test accuracy to number of trainable parameters as the criterion, models can be ranked from best to worst. Here are the top five:

```
index 0, test acc 0.872, num_params 1819690, acc/num_params 4.79203e-07,
EMBEDDING_DIM=32,HIDDEN_DIM=20,LR=0.001,OPTIM=rmsprop

index 12, test acc 0.869, num_params 1823050, acc/num_params 4.76674e-07,
EMBEDDING_DIM=32,HIDDEN_DIM=20,LR=0.001,N_LAYERS=2,OPTIM=rmsprop

index 36, test acc 0.872, num_params 1829770, acc/num_params 4.76563e-07,
DROPOUT_RATE=0.2,EMBEDDING_DIM=32,HIDDEN_DIM=20,LR=0.001,N_LAYERS=4,OPTIM=rmsprop

index 24, test acc 0.869, num_params 1826410, acc/num_params 4.75797e-07,
DROPOUT_RATE=0.2,EMBEDDING_DIM=32,HIDDEN_DIM=20,LR=0.001,N_LAYERS=3,OPTIM=rmsprop

index 6, test acc 0.859, num_params 2191750, acc/num_params 3.91924e-07,
DROPOUT_RATE=0.2,EMBEDDING_DIM=32,HIDDEN_DIM=290,LR=0.001,OPTIM=rmsprop
```

In the `diff_hparams` line, I only listed the hyperparameters that are different from the default hyperparameters.

Using **test accuracy only** as the criterion for ranking, here are the top five models:

```
acc 0.8801587496485029
diff_hparams DROPOUT_RATE=0.2,EMBEDDING_DIM=32,HIDDEN_DIM=305,LR=0.001,N_LAYERS=2,OPTIM=rmsprop

acc 0.8749008133297875
diff_hparams DROPOUT_RATE=0.2,EMBEDDING_DIM=188,HIDDEN_DIM=305,LR=0.001,OPTIM=rmsprop

acc 0.8733135115532648
diff_hparams DROPOUT_RATE=0.2,EMBEDDING_DIM=256,HIDDEN_DIM=280,LR=0.001,N_LAYERS=2,OPTIM=rmsprop

acc 0.8722222436042059
diff_hparams EMBEDDING_DIM=32,HIDDEN_DIM=20,LR=0.001,OPTIM=rmsprop

acc 0.8721230376334418
diff_hparams DROPOUT_RATE=0.2,EMBEDDING_DIM=32,HIDDEN_DIM=280,LR=0.001,N_LAYERS=2,OPTIM=rmsprop
```

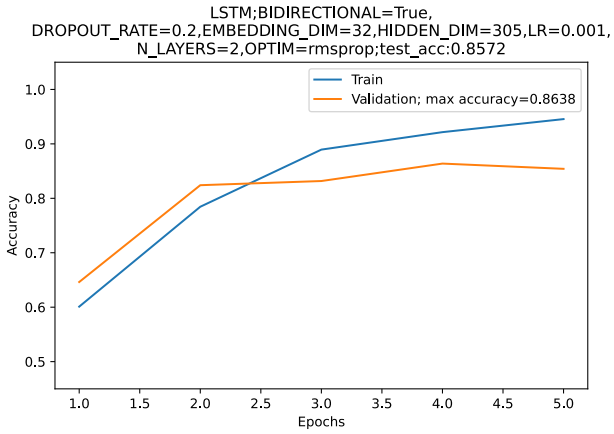
Please refer to the Jupyter Notebook for code blocks that performs compound scaling and print the text outputs. I performed a total of 48 trials with the RMSprop optimiser and $lr = 0.001$ and trained all the models for 5 epochs. This is the best settings that I found in lab 2 part (a). I sampled `N_LAYERS` from $(1, 2, 3, 4)$, `HIDDEN_DIM` from $(20, 280, 290, 305)$, and `EMBEDDING_DIM` from $(32, 180, 256)$. Although lab 2 part (c) showed that the a large number of layers led to bad models, I though that perhaps the model was going to perform well with a larger number of layers combined with larger embedding dimension and more hidden states, so I chose to test with all four values for `N_LAYERS`: $(1, 2, 3, 4)$. `HIDDEN_DIM` $\in (10, 260, 285)$ seemed to have produced good results in lab 2 part (d) (Figure 8b), so I decided to sample the region around `HIDDEN_DIM = 285` a bit more: $(20, 280, 290, 305)$. Changing embedding dimension by itself didn't seem to have had a lot of impact, so I sampled a wide range of different embedding dimensions $(32, 180, 256)$. I changed `DROPOUT_RATE` to 0.2 to most models.

Looking **only at the testing accuracy**, the model that has the highest performance using **the testing accuracy metric** performs better than all of the models that are in parts (c) to (e). The best model (testing accuracy 0.8801587) has 2976160 trainable parameters.

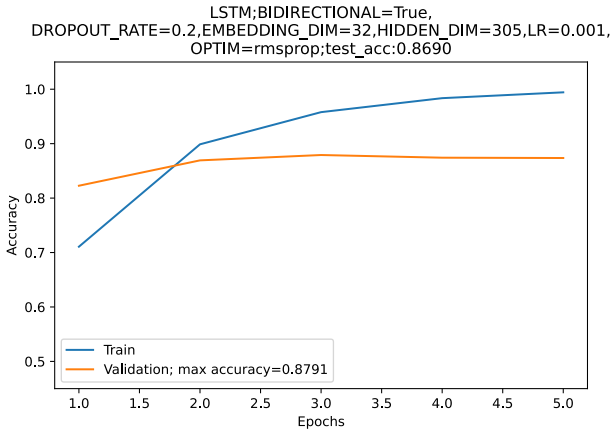
However, the best model ranked using the **ratio of testing accuracy to number of parameters** metric

has **lower test accuracy** than some models, for example the models in Figures 6g and 7e (just as examples). The test accuracy to number of parameters ratio of the "best" model found during compound scaling is worse than the model in Figure 7b, for example (test accuracy $0.8729 > 0.8720$, number of parameters $155260 < 1819690$).

(g) (Bonus, 5 pts) RNNs can be bidirectional. Use the best model discovered in (f) and make it bidirectional. Do you observe better accuracy on IMDB dataset and why?
Please see the code in the Jupyter Notebook for the modification required to make the LSTM bidirectional.



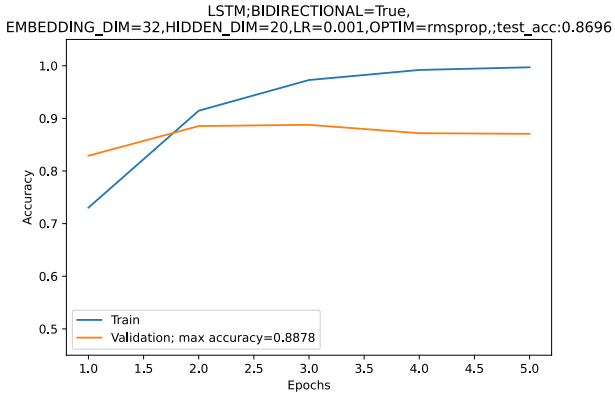
(a) Training bidirectional LSTM with best hyperparameters from part (f), 4881190 parameters.



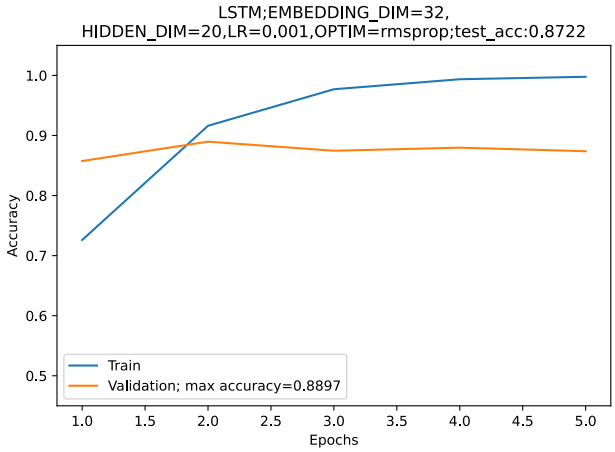
(b) Training bidirectional LSTM with best hyperparameters (except with `N_LAYERS=1`) from part (f), 2643710 parameters

Figure 9: Training with the best (or close to the best) hyperparameters, **ranked by accuracy only**.

Figure 9a shows the performance of the bidirectional model using the best hyperparameter setting found in part (f) using **the accuracy metric**.
Figure 9b shows the performance of the bidirectional model using the best hyperparameters **except** for `N_LAYERS=1` using **the accuracy metric**.
The bidirectional model with `N_LAYERS=2` (Figure 9a) seems to train more slowly compared to the bidirectional model with `N_LAYERS=1` (Figure 9b). **I do not observe better accuracy on the IMDB dataset when using bidirectional models.** Neither bidirectional LSTM model outperforms the best non-bidirectional model which had a test accuracy of 0.88. I suspect that the bidirectional model using `N_LAYERS=2` (Figure 9a) is too complex and struggles to learn quickly enough in just five epochs. The bidirectional model with `N_LAYERS=1`, which is less complex (Figure 9b), performs a lot better than the bidirectional model with `N_LAYERS=2` when trained for 5 epochs.
However, the complexity of the model is an imperfect explanation. The model that reached 0.880 test accuracy has 2976160 trainable parameters, which is more than the bidirectional model shown in Figure 9b that only has 0.8690 test accuracy. Another explanation for this could be that models that reach very high validation accuracy early on (e.g., 0.89 validation accuracy after the first epoch) suffer from over-fitting after it is trained for five epochs. This means that the testing accuracy after 5 epochs isn't necessarily the best metric for measuring the accuracy of a model and using it means that some less complex models that perform very well in the first epoch but begin to over-fit after 1-2 epochs are not considered.
Another reason for worse bidirectional model performance when using the best hyperparameters found using non-bidirectional models is that the best hyperparameters are found using non-bidirectional models and not bidirectional models. Bidirectional LSTMs may perform very well with hyperparameters that do not necessarily give the best performance for uni-directional LSTMs and vice versa. We were maximising the accuracy for non-bidirectional models, not for bidirectional models, so it's no surprise that the hyperparameters that work very well for non-bidirectional models did not work very well for bidirectional models.



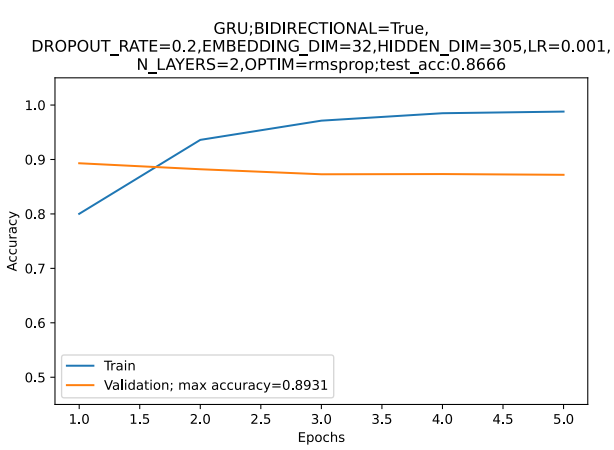
(a) Training bidirectional LSTM with best hyperparameters from part (f), 1824050 parameters.



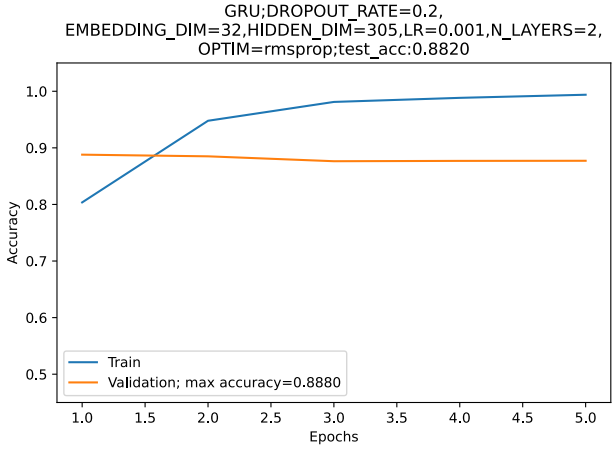
(b) Training non-bidirectional LSTM with best hyperparameters from part (f), 1819690 parameters

Figure 10: Training with the best hyperparameters, ranked by **ratio of accuracy to number of parameters**.

Figure 10 compares the non-bidirectional model (Figure 10b) to the bidirectional model (Figure 10a) using the best set of hyperparameters with the ranking criterion being the ratio of accuracy to the number of parameters. The bidirectional model (Figure 10a) has lower test accuracy and more trainable parameters, which means that the bidirectional model using the "best" hyperparameters performs worse than the non-bidirectional model.



(a) Training bidirectional GRU with best hyperparameters from part (f), 4,115,030 parameters.



(b) Training non-bidirectional GRU with best hyperparameters from part (f), 2,686,105 parameters

Figure 11: Training GRUs with the best hyperparameters, ranked by **accuracy only**.

Figure 11 compares the bidirectional and non-bidirectional GRU models using the best hyperparameters ranked by accuracy only when performing compound scaling on LSTMs. The hyperparameters used in Figure 11a are the same as the hyperparameters in Figure 9a. Again, the bidirectional model (GRU, Figure 11a) performs worse than the non-bidirectional model (GRU, Figure 11b) using both the accuracy only metric and the accuracy/num_params ratio metric.
Please refer to the Jupyter Notebook to see how I set everything up.

LabRNN

October 18, 2023

1 Implement and train a LSTM for sentiment analysis

(General Hint on Lab 1/2: Trust whatever you see from the training and report it on PDF. IDMB is far from ideal as it's more like a real-world dataset)

1.1 Step 0: set up the environment

[1]:
```
import sys
print(sys.executable)
```

```
/home/yl826/anaconda3/envs/michael/bin/python
```

[2]:
```
import functools
import sys
import numpy as np
import pandas as pd
import random
import re
import matplotlib.pyplot as plt
# import tqdm
from tqdm.auto import tqdm  # my addition
import nltk
from sklearn.model_selection import train_test_split
from nltk.corpus import stopwords
from collections import Counter
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset

from typing import Tuple, Dict, List

nltk.download('stopwords')

torch.backends.cudnn.benchmark = True

import os
os.makedirs("resources", exist_ok=True)
```

```
[nltk_data] Downloading package stopwords to /home/yl826/nltk_data…
[nltk_data]   Package stopwords is already up-to-date!
```

[3]:
```
'cuda' if torch.cuda.is_available() else 'cpu'  # my addition
```

[3]:
```
'cuda'
```

1.1.1 Hyperparameters. Do not directly touch this to mess up settings.

If you want to initalize new hyperparameter sets, use "new_hparams = HyperParams()" and change corresponding fields.

[4]:
```
class HyperParams:
    def __init__(self):
        # Constance hyperparameters. They have been tested and don't need to be␣
 ↪tuned.
        self.PAD_INDEX = 0
        self.UNK_INDEX = 1
        self.PAD_TOKEN = '<pad>'
        self.UNK_TOKEN = '<unk>'
        self.STOP_WORDS = set(stopwords.words('english'))
        self.MAX_LENGTH = 256
        self.BATCH_SIZE = 96
        self.EMBEDDING_DIM = 1
        self.HIDDEN_DIM = 100
        self.OUTPUT_DIM = 2
        self.N_LAYERS = 1
        self.DROPOUT_RATE = 0.0
        self.LR = 0.01
        self.N_EPOCHS = 5
        self.WD = 0
        self.OPTIM = "sgd"
        self.BIDIRECTIONAL = False
        self.SEED = 5


ORIG_HPARAMS = HyperParams()
```

[5]:
```
import inspect

def diff_hparams(orig_hp: HyperParams, new_hp: HyperParams) -> str:
    """
    @param orig_hp: original hyperparameters, class HyperParams
    @param new_hp: hyperparameters that have different values
    return string containing attr and vals that are different
    e.g.
    org_hyperparams = HyperParams(); new_hyperparams = HyperParams()
```

```
    new_hyperparams.N_LAYERS = 10; new_hyperparams.BIDIRECTIONAL = True
    diff_hparams(org_hyperparams, new_hyperparams)

    return 'BIDIRECTIONAL=True,N_LAYERS=10'
    """
    attr_orig = inspect.getmembers(orig_hp, lambda a:not(inspect.isroutine(a)))
    attr_orig_filtered = [a for a in attr_orig if not(a[0].startswith('__') and␣
↪a[0].endswith('__'))]

    attr_new = inspect.getmembers(new_hp, lambda a:not(inspect.isroutine(a)))
    attr_new_filtered = [a for a in attr_new if not(a[0].startswith('__') and␣
↪a[0].endswith('__'))]
    diff_hparams_dict = {}
    for idx, (n, val) in enumerate(attr_new_filtered):
        if val != attr_orig_filtered[idx][1]:
            diff_hparams_dict[n] = val
    ret = ""
    for idx, (k, v) in enumerate(diff_hparams_dict.items()):
        if type(v) == int or type(v) == float:
            ret = ret + '%s=%g,' % (k, v)
        if type(v) == bool:
            ret = ret + '%s=%r,' % (k, v)
        if type(v) == str:
            ret = ret + '%s=%s,' % (k, v)
        if idx % 4 == 0:
            ret = ret + '\n'
    return ret[0:-1]


def plot_train_val_acc(save_name, ret_dict, orig_hp, new_hp, save, gru=False):

    fig, ax = plt.subplots(1, 1)
    xx = np.linspace(1, orig_hp.N_EPOCHS, orig_hp.N_EPOCHS)

    ax.plot(xx, ret_dict['train_accs'], label='Train')
    ax.plot(xx, ret_dict['valid_accs'],
            label='Validation; max accuracy=%.4f' % (np.
↪max(ret_dict['valid_accs'])))
    ax.set_ylim([0.45, 1.05])

    diff_param = diff_hparams(orig_hp=orig_hp, new_hp=new_hp)

    lstm_or_gru = {False: 'LSTM', True: 'GRU'}

    if len(diff_param) != 0:
        title_ = lstm_or_gru[gru] + ';' + diff_param
    else:
```

```
        title_ = lstm_or_gru[gru]
    title_ += ';test_acc:%.4f' % ret_dict['test_acc']

    ax.set_xlabel('Epochs')
    ax.set_ylabel('Accuracy')
    ax.legend()
    ax.set_title(title_)
    fig.tight_layout()
    if save == 'y':
        if len(diff_param) != 0:
            plt.savefig('%s_%s.pdf' % (save_name, diff_param.strip()), dpi=500,␣
↪bbox_inches='tight')
        else:
            plt.savefig('%s_default_hp.pdf' % (save_name), dpi=500,␣
↪bbox_inches='tight')
    return
```

1.2 Lab 1(a) Implement your own data loader function.

First, you need to read the data from the dataset file on the local disk. Then, split the dataset into three sets: train, validation and test by 7:1:2 ratio. Finally return x_train, x_valid, x_test, y_train, y_valid, y_test where x represents reviews and y represent labels.

[6]:
```
from sklearn.model_selection import train_test_split  # my addition

def preprocessing_string(s):  # my addition
    return s


def load_imdb(base_csv:str = './IMDBDataset.csv'):
    """
    Load the IMDB dataset
    :param base_csv: the path of the dataset file.
    :return: train, validation and test set.
    """
    # Add your code here.
    df = pd.read_csv('IMDBDataset.csv')
    x = list(df['review'])
    y = list(df['sentiment'])
    # 7:3 train/(test&Val) ratio
    x_train, x_testVal, y_train, y_testVal = train_test_split(x, y, test_size=0.
↪3,
                                                              random_state=1002,
                                                              shuffle=True)
    # 1:2 val='train'/test ratio
    x_valid, x_test, y_valid, y_test = train_test_split(x_testVal, y_testVal,
                                                        test_size=float(2/3),
```

```
                                                    random_state=1002,
                                                    shuffle=True)
    # I'm only to run this function once, so it shouldn't matter across␣
↪different trainings if I shuffle.
    # random seed is set, so should be constant across different kernel␣
↪sessions.
    print(f'shape of train data is {len(x_train)}')
    print(f'shape of test data is {len(x_test)}')
    print(f'shape of valid data is {len(x_valid)}')
    return x_train, x_valid, x_test, y_train, y_valid, y_test
```

1.3 Lab 1(b): Implement your function to build a vocabulary based on the training corpus.

You should first compute the frequency of all the words in the training corpus. Use the given preprocess_string() function to process each word by "preprocess_string(word)". Avoid the words that are in the STOP_WORDS. Filter the words by their frequency (>=min_freq). Generate a corpus variable which contains a list of words.

[7]:
```
def build_vocab(x_train:list, min_freq: int=5, hparams=None) -> dict:
    """
    build a vocabulary based on the training corpus.
    :param x_train:  List. The training corpus. Each sample in the list is a␣
↪string of text.
    :param min_freq: Int. The frequency threshold for selecting words.
    :return: dictionary {word:index}
    """
    # Add your code here. Your code should assign corpus with a list of words.

    # split x_train into words
    words_temp = []
    for sent in x_train:
        words_lst = sent.split()
        words_lst_lower = [w.lower() for w in words_lst]
        words_temp.extend(words_lst_lower)
    # no additional string preprocessing

    corpus = Counter([w for w in words_temp if w not in hparams.STOP_WORDS])

    # sorting on the basis of most common words
    # corpus_ = sorted(corpus, key=corpus.get, reverse=True)[:1000]
    corpus_ = [word for word, freq in corpus.items() if freq >= min_freq]
    # creating a dict
    vocab = {word : idx + 2 for idx, word in enumerate(corpus_)}
    vocab[hparams.PAD_TOKEN] = hparams.PAD_INDEX
    vocab[hparams.UNK_TOKEN] = hparams.UNK_INDEX
    return vocab
```

```
# vocab = build_vocab(x_train, hparams=hparams)
```

1.4 Lab 1(c): Implement your tokenize function.

You should leverage the given preprocess_string() function to process each word by "preprocess_string(word)". For each word, find its index in the vocabulary. Return a list of int that represents the indices of words in the example.

[8]:
```
def tokenize(vocab: dict, example: str) -> list:
    """
    Tokenize the give example string into a list of token indices.
    :param vocab: dict, the vocabulary.
    :param example: a string of text.
    :return: a list of token indices.
    """
    # Your code here.
    ex_tok = example.split()
    ex = [w.lower() for w in ex_tok]
    # return [vocab[w] if w in vocab else vocab[ORIG_HPARAMS.UNK_TOKEN] for w␣
 ↪in ex]
    return [vocab[w] for w in ex if w in vocab]


# for index, (word, idx) in enumerate(vocab.items()):
#     print(word, idx)
#     if index > 10:
#         break
# example_tokens = tokenize(vocab=vocab, example="This probably the worst ever␣
 ↪movie made."
#                           " Roger Corman the best")
# print(example_tokens)
```

1.5 Lab 1 (d): Implement the getitem function. Given an index i, you should return the i-th review and label.

The review is originally a string. Please tokenize it into a sequence of token indices. Use the max_length parameter to truncate the sequence so that it contains at most max_length tokens. Convert the label string ('positive'/'negative') to a binary index. 'positive' is 1 and 'negative' is 0. Return a dictionary containing three keys: 'ids', 'length', 'label' which represent the list of token ids, the length of the sequence, the binary label.

[9]:
```
class IMDB(Dataset):
    def __init__(self, x, y, vocab, max_length=256) -> None:
        """
        :param x: list of reviews
```

```
        :param y: list of labels
        :param vocab: vocabulary dictionary {word:index}.
        :param max_length: the maximum sequence length.
        """
        self.x = x
        self.y = y
        self.vocab = vocab
        self.max_length = max_length

    def __getitem__(self, idx: int) -> Dict:
        """
        Return the tokenized review and label by the given index.
        :param idx: index of the sample.
        :return: a dictionary containing three keys: 'ids', 'length',
        'label' which represent the list of token ids,
        the length of the sequence, the binary label.
        """
        # Add your code here.
        pn_dict = {'positive': 1, 'negative': 0}

        review, binary_label = tokenize(vocab=self.vocab, example=self.x[idx]),␣
↪pn_dict[self.y[idx]]
        # print(len(review), review[0:2])
        if len(review) > self.max_length:
            return {'ids': review[0:self.max_length], 'length': self.
↪max_length, 'label': binary_label}
        else:
            return {'ids': review, 'length': len(review), 'label': binary_label}

    def __len__(self) -> int:
        return len(self.x)


def collate(batch, pad_index):
    batch_ids = [torch.LongTensor(i['ids']) for i in batch]
    batch_ids = nn.utils.rnn.pad_sequence(batch_ids, padding_value=pad_index,␣
↪batch_first=True)
    batch_length = torch.Tensor([i['length'] for i in batch])
    batch_label = torch.LongTensor([i['label'] for i in batch])
    batch = {'ids': batch_ids, 'length': batch_length, 'label': batch_label}
    return batch


# imdb_obj = IMDB(x=x_train, y=y_train, vocab=vocab)
# print(imdb_obj[2])
collate_fn = collate
```

1.6 Lab 1 (e): Implement the LSTM model for sentiment analysis.

Q(a): Implement the initialization function. Your task is to create the model by stacking several necessary layers including an embedding layer, a lstm cell, a linear layer, and a dropout layer. You can call functions from Pytorch's nn library. For example, nn.Embedding, nn.LSTM, nn.Linear. Q(b): Implement the forward function. Decide where to apply dropout. The sequences in the batch have different lengths. Write/call a function to pad the sequences into the same length. Apply a fully-connected (fc) layer to the output of the LSTM layer. Return the output features which is of size [batch size, output dim].

[10]:
```
def init_weights(m):
    if isinstance(m, nn.Embedding):
        nn.init.xavier_normal_(m.weight)
    elif isinstance(m, nn.Linear):
        nn.init.xavier_normal_(m.weight)
        nn.init.zeros_(m.bias)
    elif isinstance(m, nn.LSTM) or isinstance(m, nn.GRU):
        for name, param in m.named_parameters():
            if 'bias' in name:
                nn.init.zeros_(param)
            elif 'weight' in name:
                nn.init.orthogonal_(param)

class LSTM(nn.Module):
    def __init__(
        self,
        vocab_size: int,
        embedding_dim: int,
        hidden_dim: int,
        output_dim: int,
        n_layers: int,
        dropout_rate: float,
        pad_index: int,
        bidirectional: bool,
        **kwargs):
        """
        Create a LSTM model for classification.
        :param vocab_size: size of the vocabulary
        :param embedding_dim: dimension of embeddings
        :param hidden_dim: dimension of hidden features
        :param output_dim: dimension of the output layer which equals to the␣
 ↪number of labels.
        :param n_layers: number of layers.
        :param dropout_rate: dropout rate.
        :param pad_index: index of the padding token.we
        """
        super().__init__()
        # Add your code here. Initializing each layer by the given arguments.
```

```
        self.pad_index = pad_index
        self.hidden_dim = hidden_dim
        self.embedding = nn.Embedding(num_embeddings=vocab_size,
                                      embedding_dim=embedding_dim,␣
↪padding_idx=pad_index,
                                      max_norm=None, norm_type=2.0,␣
↪scale_grad_by_freq=False,
                                      sparse=False, _weight=None, _freeze=False,
                                      device=None, dtype=None)
        lstm_dropout = None
        if n_layers > 1:
            lstm_dropout = dropout_rate
        else:
            lstm_dropout = 0.0
        self.lstm = nn.LSTM(input_size=embedding_dim, hidden_size=hidden_dim,
                            num_layers=n_layers, bias=True, batch_first=True,
                            dropout=lstm_dropout, bidirectional=bidirectional,␣
↪proj_size=0,
                            device=None, dtype=None)
        # dropout=0.0 in self.lstm since n_layers is only 1 and the last lstm␣
↪layer
        # doesn't have any dropout
        # batch_first=True since ids is batch_size * sequence_len

        self.dropout = nn.Dropout(dropout_rate)

        self.fc = nn.Linear(in_features=hidden_dim, out_features=output_dim,
                            bias=True, device=None, dtype=None)
        if bidirectional: # bidirectional: 2*hidden_dim is needed
            # since lstm output is double the size
            self.fc = nn.Linear(in_features=2*hidden_dim,␣
↪out_features=output_dim,
                                bias=True, device=None, dtype=None)
        # Weight initialization. DO NOT CHANGE!
        if "weight_init_fn" not in kwargs:
            self.apply(init_weights)
        else:
            self.apply(kwargs["weight_init_fn"])


    def forward(self, ids:torch.Tensor, length:torch.Tensor):
        """
        Feed the given token ids to the model.
        :param ids: [batch size, seq len] batch of token ids.
        :param length: [batch size] batch of length of the token ids.
        :return: prediction of size [batch size, output dim].
        """
```

```
        # note: using batch_first=True
        # do we need to pad? the dataloader seems to be pretty good at padding
        # NO NEED to use nn.utils.rnn.pad_sequence since collate_fn already␣
↪pads everything
        # and each batch has already been padded.
        padded_ids = nn.utils.rnn.pad_sequence(sequences=ids, batch_first=True,
                                               padding_value=self.pad_index)

        # Add your code here.
        out = self.embedding(padded_ids)
        # packs the embeddings (better for computation)
        out = nn.utils.rnn.pack_padded_sequence(out, length, batch_first=True,␣
↪enforce_sorted=False)

        # Pass embedded input through the LSTM and dropout layers
        out, hidden = self.lstm(out)
        # otuput of lstm is packed sequence
        out, _ = nn.utils.rnn.pad_packed_sequence(out, batch_first=True,
                                                  padding_value=self.pad_index)
        # unpack the packed thing, should result in tensor of shape (batch␣
↪size, seq len, hidden_size)
        # link to explain the range indexing below: take out every length-1-th␣
↪element for each batch
        # https://stackoverflow.com/questions/53123009/
↪using-python-range-objects-to-index-into-numpy-arrays
        out = self.dropout(out[range(out.shape[0]), length.int() - 1, :])
        # dropout(tensor of shape [batch_size, hidden_dim]) i.e. apply dropout␣
↪on
        # last hidden state of each input in the batch

        prediction = self.fc(out)
        # Get the output from the last time step
        return prediction
```

[11]:
```
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)


def train(dataloader, model, criterion, optimizer, scheduler, device):
    model.train()
    epoch_losses = []
    epoch_accs = []
    # orig: tqdm.tqdm(dataloader, desc='training...', file=sys.stdout) with␣
↪import tqdm
    # for batch in tqdm(dataloader, desc='training...', file=sys.stdout,␣
↪miniters=int(223265/100)):
```

```
    for batch in tqdm(dataloader, desc='training...', file=sys.stdout,␣
 ↪disable=True):
        # my addition
        ids = batch['ids'].to(device)
        length = batch['length']
        label = batch['label'].to(device)
        prediction = model(ids, length)
        loss = criterion(prediction, label)
        accuracy = get_accuracy(prediction, label)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        epoch_losses.append(loss.item())
        epoch_accs.append(accuracy.item())
        scheduler.step()

    return epoch_losses, epoch_accs

def evaluate(dataloader, model, criterion, device):
    model.eval()
    epoch_losses = []
    epoch_accs = []

    with torch.no_grad():
        # orig: tqdm.tqdm(dataloader, desc='training...', file=sys.stdout) with␣
 ↪import tqdm
        # for batch in tqdm(dataloader, desc='evaluating...', file=sys.stdout,␣
 ↪miniters=int(223265/100)):
        for batch in tqdm(dataloader, desc='evaluating...', file=sys.stdout,␣
 ↪disable=True):
            # my addition
            ids = batch['ids'].to(device)
            length = batch['length']
            label = batch['label'].to(device)
            prediction = model(ids, length)
            loss = criterion(prediction, label)
            accuracy = get_accuracy(prediction, label)
            epoch_losses.append(loss.item())
            epoch_accs.append(accuracy.item())

    return epoch_losses, epoch_accs

def get_accuracy(prediction, label):
    batch_size, _ = prediction.shape
    predicted_classes = prediction.argmax(dim=-1)
    correct_predictions = predicted_classes.eq(label).sum()
    accuracy = correct_predictions / batch_size
```

```
    return accuracy

def predict_sentiment(text, model, vocab, device):
    tokens = tokenize(vocab, text)
    ids = [vocab[t] if t in vocab else UNK_INDEX for t in tokens]
    length = torch.LongTensor([len(ids)])
    tensor = torch.LongTensor(ids).unsqueeze(dim=0).to(device)
    prediction = model(tensor, length).squeeze(dim=0)
    probability = torch.softmax(prediction, dim=-1)
    predicted_class = prediction.argmax(dim=-1).item()
    predicted_probability = probability[predicted_class].item()
    return predicted_class, predicted_probability
```

1.6.1 Lab 1 (g) Implement GRU.

[12]:
```
class GRU(nn.Module):
    def __init__(
        self,
        vocab_size: int,
        embedding_dim: int,
        hidden_dim: int,
        output_dim: int,
        n_layers: int,
        dropout_rate: float,
        pad_index: int,
        bidirectional: bool,
        **kwargs):
        """
        Create a LSTM model for classification.
        :param vocab_size: size of the vocabulary
        :param embedding_dim: dimension of embeddings
        :param hidden_dim: dimension of hidden features
        :param output_dim: dimension of the output layer which equals to the␣
 ↪number of labels.
        :param n_layers: number of layers.
        :param dropout_rate: dropout rate.
        :param pad_index: index of the padding token.we
        """
        super().__init__()
        # Add your code here. Initializing each layer by the given arguments.
        self.pad_index = pad_index
        self.embedding = nn.Embedding(num_embeddings=vocab_size,
                                      embedding_dim=embedding_dim,␣
 ↪padding_idx=pad_index,
                                      max_norm=None, norm_type=2.0,␣
 ↪scale_grad_by_freq=False,
```

```
                                    sparse=False, _weight=None, _freeze=False,
                                    device=None, dtype=None)
        gru_dropout = None
        if n_layers > 1:
            gru_dropout = dropout_rate
        else:
            gru_dropout = 0.0

        self.gru = nn.GRU(input_size=embedding_dim, hidden_size=hidden_dim,
                          num_layers=n_layers, bias=True, batch_first=True,
                          dropout=gru_dropout, bidirectional=bidirectional,
                          device=None, dtype=None)

        self.fc = nn.Linear(in_features=hidden_dim, out_features=output_dim,
                            bias=True, device=None, dtype=None)
        if bidirectional:  # hidden_dim*2, same reason as in LSTM
            self.fc = nn.Linear(in_features=hidden_dim*2,␣
↪out_features=output_dim,
                                bias=True, device=None, dtype=None)
        self.dropout = nn.Dropout(dropout_rate)

        # Weight Initialization. DO NOT CHANGE!
        if "weight_init_fn" not in kwargs:
            self.apply(init_weights)
        else:
            self.apply(kwargs["weight_init_fn"])


    def forward(self, ids:torch.Tensor, length:torch.Tensor):
        """
        Feed the given token ids to the model.
        :param ids: [batch size, seq len] batch of token ids.
        :param length: [batch size] batch of length of the token ids.
        :return: prediction of size [batch size, output dim].
        """
        # Add your code here.

        # DO NOT need to use nn.utils.rnn.pad_sequence since collate_fn already␣
↪pads everything
        # and each batch has already been padded.
        padded_ids = nn.utils.rnn.pad_sequence(sequences=ids, batch_first=True,
                                               padding_value=self.pad_index)
        out = self.embedding(padded_ids)
        out = nn.utils.rnn.pack_padded_sequence(out, length, batch_first=True,␣
↪enforce_sorted=False)

        out, _ = self.gru(out)  # second output is hidden state
```

```
        # unpack gru_out since it's packed, results in tensor of shape (batch␣
↪size, seq len, hidden_size)
        out, _ = nn.utils.rnn.pad_packed_sequence(out, batch_first=True,
                                                  padding_value=self.pad_index)

        # Apply dropout to the GRU output
        out = self.dropout(out[range(out.shape[0]), length.int() - 1, :])
        # array indexing:
        # for each input, take the last non-padded hidden state (which is where␣
↪length comes in).
        # ends up with tensor of shape [batch size, hidden_dim]

        prediction = self.fc(out)
        return prediction
```

1.6.2 Learning rate warmup. DO NOT TOUCH!

[13]:
```
class ConstantWithWarmup(torch.optim.lr_scheduler._LRScheduler):
    def __init__(
        self,
        optimizer,
        num_warmup_steps: int,
    ):
        self.num_warmup_steps = num_warmup_steps
        super().__init__(optimizer)

    def get_lr(self):
        if self._step_count <= self.num_warmup_steps:
            # warmup
            scale = 1.0 - (self.num_warmup_steps - self._step_count) / self.
↪num_warmup_steps
            lr = [base_lr * scale for base_lr in self.base_lrs]
            self.last_lr = lr
        else:
            lr = self.base_lrs
        return lr
```

1.6.3 Implement the training / validation iteration here.

[14]:
```
CHECKPOINT_FOLDER = "./saved_model/"
# computes x_train, x_valid... etc once only for all models
x_train, x_valid, x_test, y_train, y_valid, y_test = load_imdb()


def train_and_test_model_with_hparams(hparams, model_type="lstm", **kwargs):
```

```
    # Seeding. DO NOT TOUCH! DO NOT TOUCH hparams.SEED!
    # Set the random seeds.
    torch.manual_seed(hparams.SEED)
    random.seed(hparams.SEED)
    np.random.seed(hparams.SEED)

    vocab = build_vocab(x_train, hparams=hparams)

    vocab_size = len(vocab)
    print(f'Length of vocabulary is {vocab_size}')
    # use x_train, x_valid, x_test, y_train, y_valid, y_test as global
    # better for comparing betw models.
    train_data = IMDB(x_train, y_train, vocab, hparams.MAX_LENGTH)
    valid_data = IMDB(x_valid, y_valid, vocab, hparams.MAX_LENGTH)
    test_data = IMDB(x_test, y_test, vocab, hparams.MAX_LENGTH)

    collate = functools.partial(collate_fn, pad_index=hparams.PAD_INDEX)

    train_dataloader = torch.utils.data.DataLoader(
        train_data, batch_size=hparams.BATCH_SIZE, collate_fn=collate,␣
↪shuffle=True)
    valid_dataloader = torch.utils.data.DataLoader(
        valid_data, batch_size=hparams.BATCH_SIZE, collate_fn=collate)
    test_dataloader = torch.utils.data.DataLoader(
        test_data, batch_size=hparams.BATCH_SIZE, collate_fn=collate)

    # Model
    if "override_models_with_gru" in kwargs and␣
↪kwargs["override_models_with_gru"]:
        model = GRU(
            vocab_size,
            hparams.EMBEDDING_DIM,
            hparams.HIDDEN_DIM,
            hparams.OUTPUT_DIM,
            hparams.N_LAYERS,
            hparams.DROPOUT_RATE,
            hparams.PAD_INDEX,
            hparams.BIDIRECTIONAL,
            **kwargs)
    else:
        model = LSTM(
            vocab_size,
            hparams.EMBEDDING_DIM,
            hparams.HIDDEN_DIM,
            hparams.OUTPUT_DIM,
            hparams.N_LAYERS,
            hparams.DROPOUT_RATE,
```

```
            hparams.PAD_INDEX,
            hparams.BIDIRECTIONAL,
            **kwargs)
    num_params = count_parameters(model)
    print(f'The model has {num_params:,} trainable parameters')


    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    model = model.to(device)

    # Optimization. Lab 2 (a)(b) should choose one of them.
    # DO NOT TOUCH optimizer-specific hyperparameters! (e.g., eps, momentum)
    # DO NOT change optimizer implementations!
    if hparams.OPTIM == "sgd":
        optimizer = optim.SGD(
            model.parameters(), lr=hparams.LR, weight_decay=hparams.WD,␣
↪momentum=.9)
    elif hparams.OPTIM == "adagrad":
        optimizer = optim.Adagrad(
            model.parameters(), lr=hparams.LR, weight_decay=hparams.WD,␣
↪eps=1e-6)
    elif hparams.OPTIM == "adam":
        optimizer = optim.Adam(
            model.parameters(), lr=hparams.LR, weight_decay=hparams.WD,␣
↪eps=1e-6)
    elif hparams.OPTIM == "rmsprop":
        optimizer = optim.RMSprop(
            model.parameters(), lr=hparams.LR, weight_decay=hparams.WD,␣
↪eps=1e-6, momentum=.9)
    else:
        raise NotImplementedError("Optimizer not implemented!")

    criterion = nn.CrossEntropyLoss()
    criterion = criterion.to(device)

    # Start training
    best_valid_loss = float('inf')
    train_losses = []
    train_accs = []
    valid_losses = []
    valid_accs = []

    # Warmup Scheduler. DO NOT TOUCH!
    WARMUP_STEPS = 200
    lr_scheduler = ConstantWithWarmup(optimizer, WARMUP_STEPS)

    for epoch in range(hparams.N_EPOCHS):
```

```
        # Your code: implement the training process and save the best model.

        train_loss, train_acc = train(dataloader=train_dataloader, model=model,
                                      criterion=criterion, optimizer=optimizer,
                                      scheduler=lr_scheduler, device=device)

        valid_loss, valid_acc = evaluate(dataloader=valid_dataloader,␣
↪model=model,
                                         criterion=criterion, device=device)

        epoch_train_loss = np.mean(train_loss)
        epoch_train_acc = np.mean(train_acc)
        epoch_valid_loss = np.mean(valid_loss)
        epoch_valid_acc = np.mean(valid_acc)

        # my addition
        train_losses.append(epoch_train_loss)
        train_accs.append(epoch_train_acc)
        valid_losses.append(epoch_valid_loss)
        valid_accs.append(epoch_valid_acc)

        # Save the model that achieves the smallest validation loss.
        if epoch_valid_loss < best_valid_loss:
            best_val_acc = epoch_valid_acc
            if not os.path.exists(CHECKPOINT_FOLDER):
               os.makedirs(CHECKPOINT_FOLDER)
            print("Saving ...")
            state = {'state_dict': model.state_dict(),
                     'epoch': epoch}
            torch.save(state, os.path.join(CHECKPOINT_FOLDER, 'hw3_LSTM.pth'))
            # Your code: save the best model somewhere (no need to submit it to␣
↪Sakai)

        print(f'epoch: {epoch+1}')
        print(f'train_loss: {epoch_train_loss:.3f}, train_acc: {epoch_train_acc:
↪.3f}')
        print(f'valid_loss: {epoch_valid_loss:.3f}, valid_acc: {epoch_valid_acc:
↪.3f}')


    # Your Code: Load the best model's weights.

    state_dict = torch.load(os.path.join(CHECKPOINT_FOLDER + 'hw3_LSTM.
↪pth'))['state_dict']
    model.load_state_dict(state_dict)
    model = model.cuda()
```

```
    # Your Code: evaluate test loss on testing dataset (NOT Validation)
    test_loss, test_acc = evaluate(dataloader=test_dataloader, model=model,
                                   criterion=criterion, device=device)

    epoch_test_loss = np.mean(test_loss)
    epoch_test_acc = np.mean(test_acc)
    print(f'test_loss: {epoch_test_loss:.3f}, test_acc: {epoch_test_acc:.3f}')

    # Free memory for later usage.
    del model
    torch.cuda.empty_cache()
    return {
        'num_params': num_params,
        "test_loss": epoch_test_loss,
        "test_acc": epoch_test_acc,
        "train_loss": train_losses,
        "train_accs": train_accs,
        "valid_loss": valid_losses,
        "valid_accs": valid_accs
    }
```

```
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
```

1.6.4 Lab 1 (f): Train model with original hyperparameters, for LSTM.

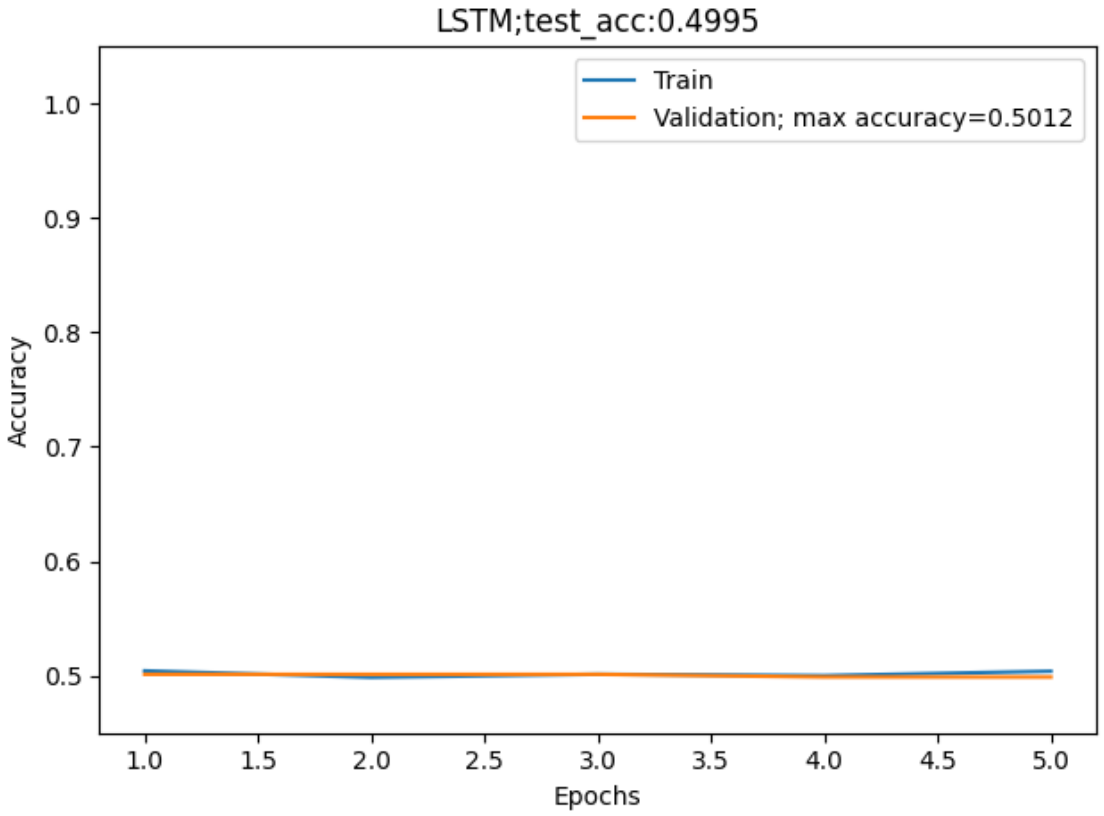
Train the model with default hyperparameter settings.

[15]:
```
org_hyperparams = HyperParams()
rd_1f = train_and_test_model_with_hparams(org_hyperparams,␣
 ↪"lstm_1layer_base_sgd_e32_h100")
```

```
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 98,131 trainable parameters
Saving …
epoch: 1
train_loss: 0.693, train_acc: 0.504
valid_loss: 0.693, valid_acc: 0.501
Saving …
epoch: 2
train_loss: 0.693, train_acc: 0.499
valid_loss: 0.693, valid_acc: 0.501
Saving …
```

```
epoch: 3
train_loss: 0.693, train_acc: 0.501
valid_loss: 0.694, valid_acc: 0.501
Saving …
epoch: 4
train_loss: 0.693, train_acc: 0.500
valid_loss: 0.693, valid_acc: 0.499
Saving …
epoch: 5
train_loss: 0.693, train_acc: 0.504
valid_loss: 0.694, valid_acc: 0.499
test_loss: 0.694, test_acc: 0.500
```

[16]:
```
plot_train_val_acc(save_name='q1f', ret_dict=rd_1f, orig_hp=ORIG_HPARAMS,
                   new_hp=org_hyperparams, save='y', gru=False)
```

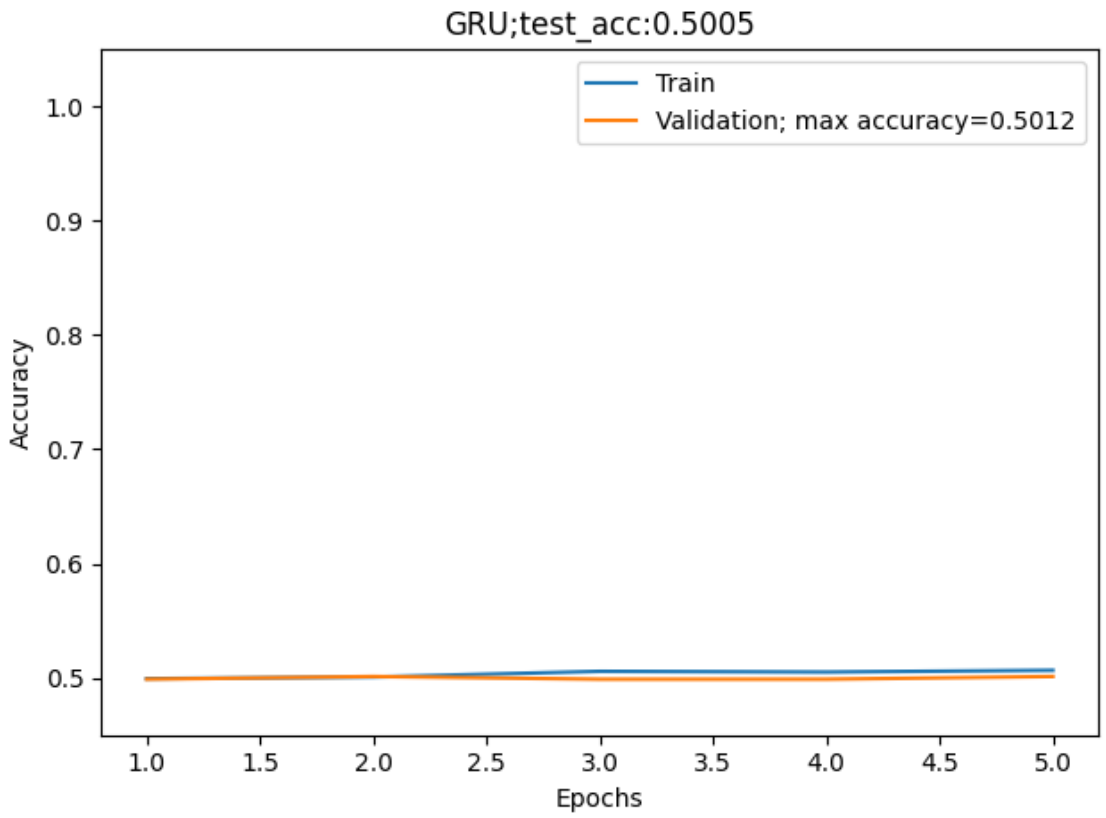


1.6.5 Lag 1 (h) Train GRU with vanilla hyperparameters.

[17]:
```
org_hyperparams = HyperParams()
rd_1h = train_and_test_model_with_hparams(org_hyperparams,␣
 ↪"gru_1layer_base_sgd_e32_h100",
                                           override_models_with_gru=True)
```

```
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 87,831 trainable parameters
Saving …
epoch: 1
train_loss: 0.694, train_acc: 0.499
valid_loss: 0.694, valid_acc: 0.499
Saving …
epoch: 2
train_loss: 0.694, train_acc: 0.501
valid_loss: 0.693, valid_acc: 0.501
Saving …
epoch: 3
train_loss: 0.693, train_acc: 0.506
valid_loss: 0.694, valid_acc: 0.499
Saving …
epoch: 4
train_loss: 0.693, train_acc: 0.505
valid_loss: 0.694, valid_acc: 0.499
Saving …
epoch: 5
train_loss: 0.693, train_acc: 0.507
valid_loss: 0.693, valid_acc: 0.501
test_loss: 0.693, test_acc: 0.500
```

[18]:
```
plot_train_val_acc(save_name='q1h', ret_dict=rd_1h, orig_hp=ORIG_HPARAMS,
                   new_hp=org_hyperparams, save='y', gru=True)
```



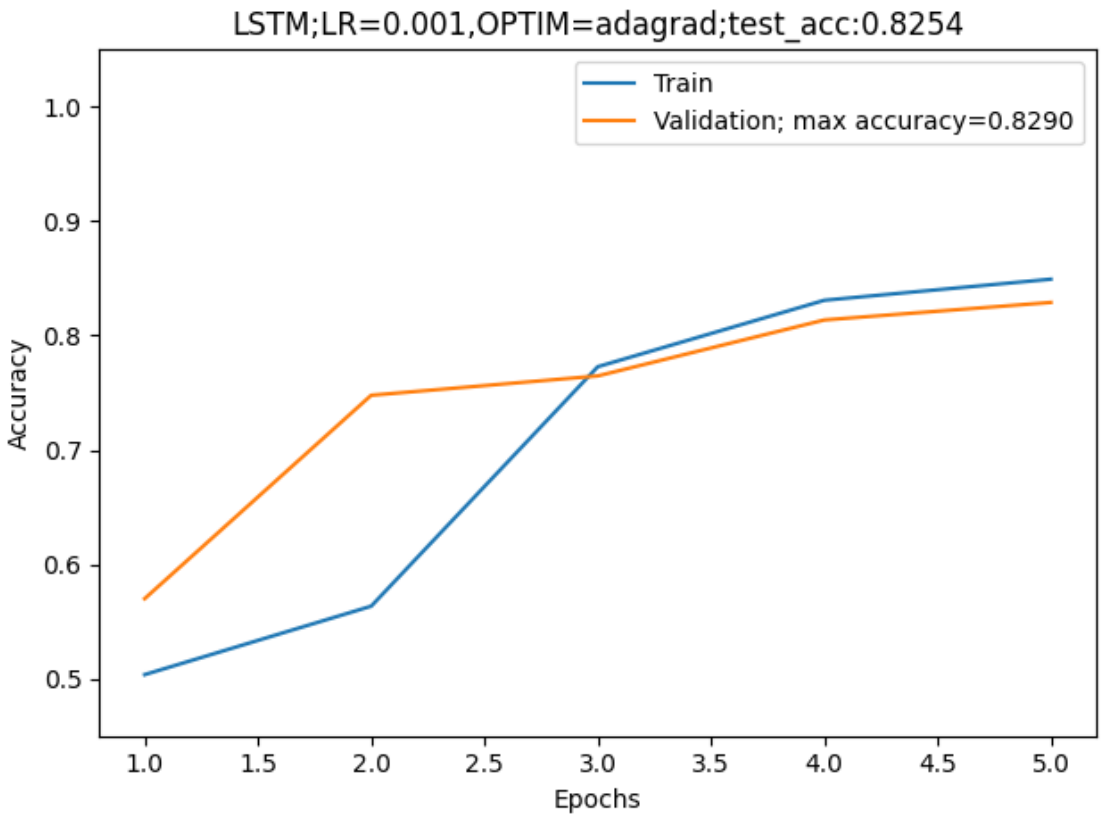
1.6.6 Lab 2 (a) Study of LSTM Optimizers. Hint: For adaptive optimizers, we recommend using a learning rate of 0.001 (instead of 0.01).

[19]:
```
q2a_adagrad_hparams = HyperParams()
q2a_adagrad_hparams.LR = 0.001
q2a_adagrad_hparams.OPTIM = 'adagrad'
rd_2a_adagrad = train_and_test_model_with_hparams(hparams=q2a_adagrad_hparams,␣
↪model_type="lstm")
```

```
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 98,131 trainable parameters
Saving …
epoch: 1
train_loss: 0.693, train_acc: 0.504
valid_loss: 0.693, valid_acc: 0.570
Saving …
epoch: 2
train_loss: 0.690, train_acc: 0.563
```

```
valid_loss: 0.636, valid_acc: 0.748
Saving …
epoch: 3
train_loss: 0.587, train_acc: 0.773
valid_loss: 0.592, valid_acc: 0.765
Saving …
epoch: 4
train_loss: 0.514, train_acc: 0.831
valid_loss: 0.535, valid_acc: 0.814
Saving …
epoch: 5
train_loss: 0.481, train_acc: 0.849
valid_loss: 0.510, valid_acc: 0.829
test_loss: 0.506, test_acc: 0.825
```

[20]:
```
plot_train_val_acc(save_name='q2a_adagrad', ret_dict=rd_2a_adagrad,␣
↪orig_hp=ORIG_HPARAMS,
                   new_hp=q2a_adagrad_hparams, save='y', gru=False)
```

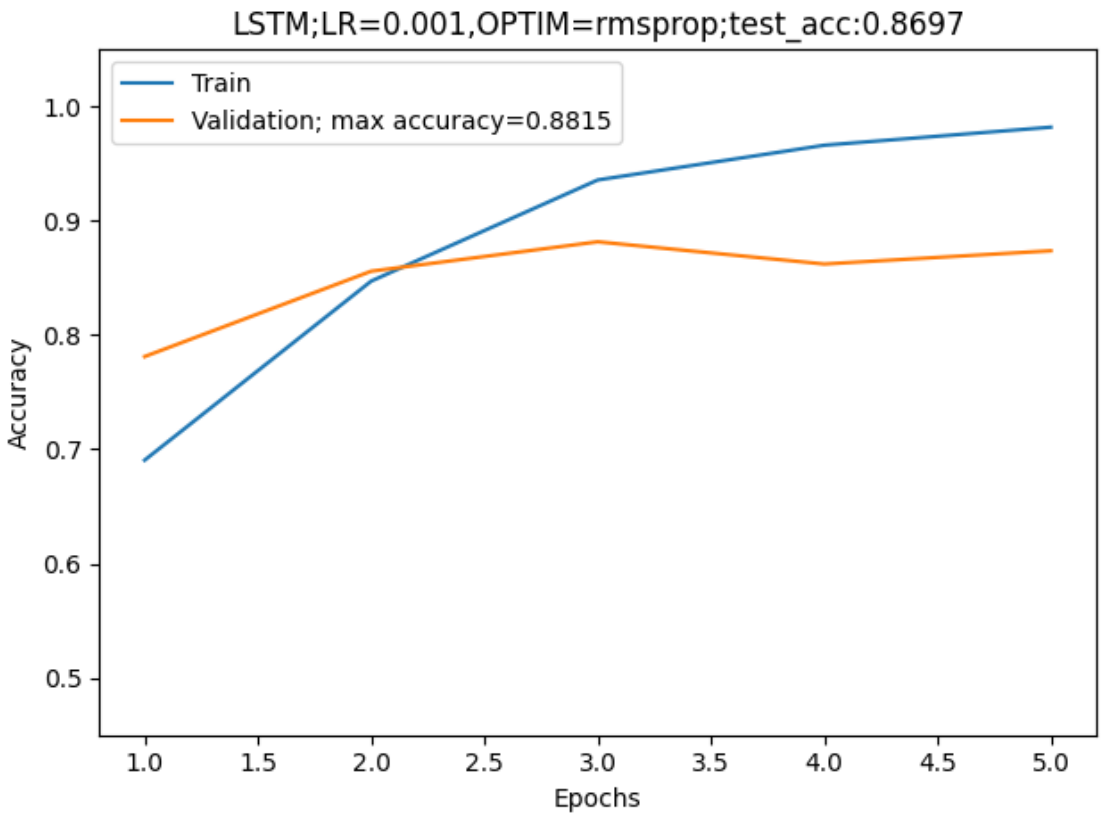


[21]:
```
q2a_rmsprop_hparams = HyperParams()
q2a_rmsprop_hparams.LR = 0.001
```

```
q2a_rmsprop_hparams.OPTIM = 'rmsprop'
rd_2a_rmsprop = train_and_test_model_with_hparams(hparams=q2a_rmsprop_hparams,␣
↪model_type="lstm")
```

```
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 98,131 trainable parameters
Saving …
epoch: 1
train_loss: 0.583, train_acc: 0.690
valid_loss: 0.468, valid_acc: 0.781
Saving …
epoch: 2
train_loss: 0.354, train_acc: 0.847
valid_loss: 0.349, valid_acc: 0.856
Saving …
epoch: 3
train_loss: 0.175, train_acc: 0.936
valid_loss: 0.296, valid_acc: 0.881
Saving …
epoch: 4
train_loss: 0.101, train_acc: 0.966
valid_loss: 0.350, valid_acc: 0.862
Saving …
epoch: 5
train_loss: 0.058, train_acc: 0.982
valid_loss: 0.406, valid_acc: 0.874
test_loss: 0.438, test_acc: 0.870
```

[22]:
```
plot_train_val_acc(save_name='q2a_rmsprop', ret_dict=rd_2a_rmsprop,␣
↪orig_hp=ORIG_HPARAMS,
                   new_hp=q2a_rmsprop_hparams, save='y', gru=False)
```

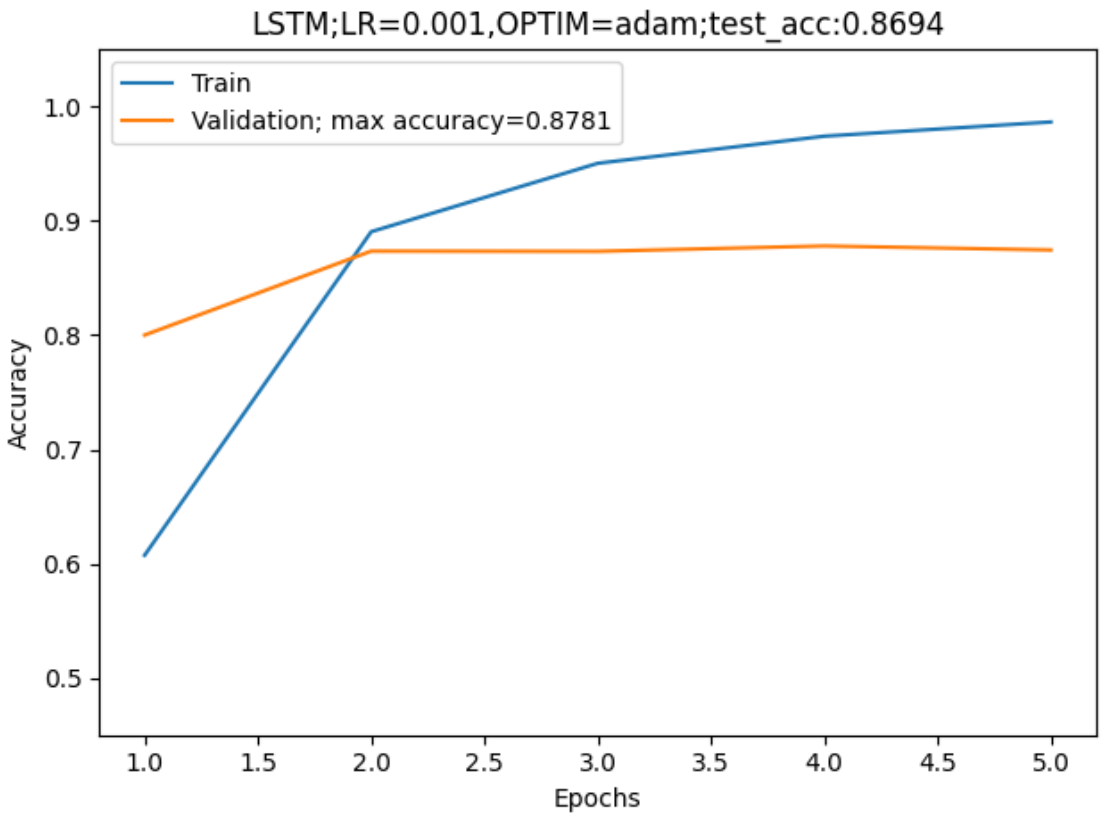


[23]:
```
q2a_adam_hparams = HyperParams()
q2a_adam_hparams.LR = 0.001
q2a_adam_hparams.OPTIM = 'adam'
rd_2a_adam = train_and_test_model_with_hparams(hparams=q2a_adam_hparams,␣
 ↪model_type="lstm")
```

```
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 98,131 trainable parameters
Saving …
epoch: 1
train_loss: 0.633, train_acc: 0.607
valid_loss: 0.417, valid_acc: 0.800
Saving …
epoch: 2
train_loss: 0.273, train_acc: 0.891
valid_loss: 0.293, valid_acc: 0.874
Saving …
epoch: 3
```

```
train_loss: 0.144, train_acc: 0.950
valid_loss: 0.294, valid_acc: 0.873
Saving …
epoch: 4
train_loss: 0.082, train_acc: 0.974
valid_loss: 0.335, valid_acc: 0.878
Saving …
epoch: 5
train_loss: 0.048, train_acc: 0.987
valid_loss: 0.416, valid_acc: 0.875
test_loss: 0.444, test_acc: 0.869
```

[24]:
```
plot_train_val_acc(save_name='q2a_adam', ret_dict=rd_2a_adam,␣
↪orig_hp=ORIG_HPARAMS,
                   new_hp=q2a_adam_hparams, save='y', gru=False)
```

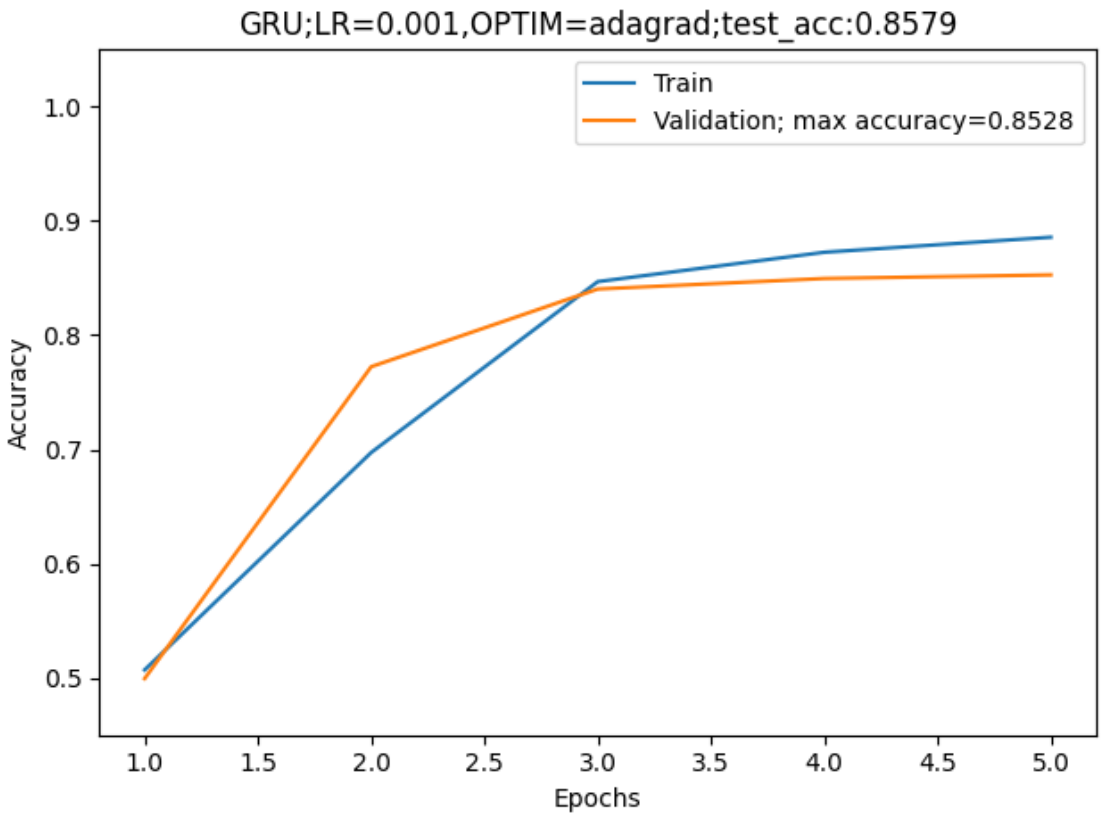


1.6.7 Lab 2 (b): Study of GRU Optimizers. Hint: For adaptive optimizers, we recommend using a learning rate of 0.001 (instead of 0.01).

[25]:
```
q2b_adagrad_hparams = HyperParams()
q2b_adagrad_hparams.LR = 0.001
q2b_adagrad_hparams.OPTIM = 'adagrad'
rd_2b_adagrad = train_and_test_model_with_hparams(q2b_adagrad_hparams, "gru",
                                                  override_models_with_gru=True)
```

```
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 87,831 trainable parameters
Saving …
epoch: 1
train_loss: 0.693, train_acc: 0.507
valid_loss: 0.693, valid_acc: 0.499
Saving …
epoch: 2
train_loss: 0.629, train_acc: 0.697
valid_loss: 0.587, valid_acc: 0.772
Saving …
epoch: 3
train_loss: 0.496, train_acc: 0.847
valid_loss: 0.489, valid_acc: 0.840
Saving …
epoch: 4
train_loss: 0.434, train_acc: 0.873
valid_loss: 0.466, valid_acc: 0.850
Saving …
epoch: 5
train_loss: 0.390, train_acc: 0.886
valid_loss: 0.438, valid_acc: 0.853
test_loss: 0.430, test_acc: 0.858
```

[26]:
```
plot_train_val_acc(save_name='q2b_adagrad', ret_dict=rd_2b_adagrad,␣
↪orig_hp=ORIG_HPARAMS,
                   new_hp=q2b_adagrad_hparams, save='y', gru=True)
```

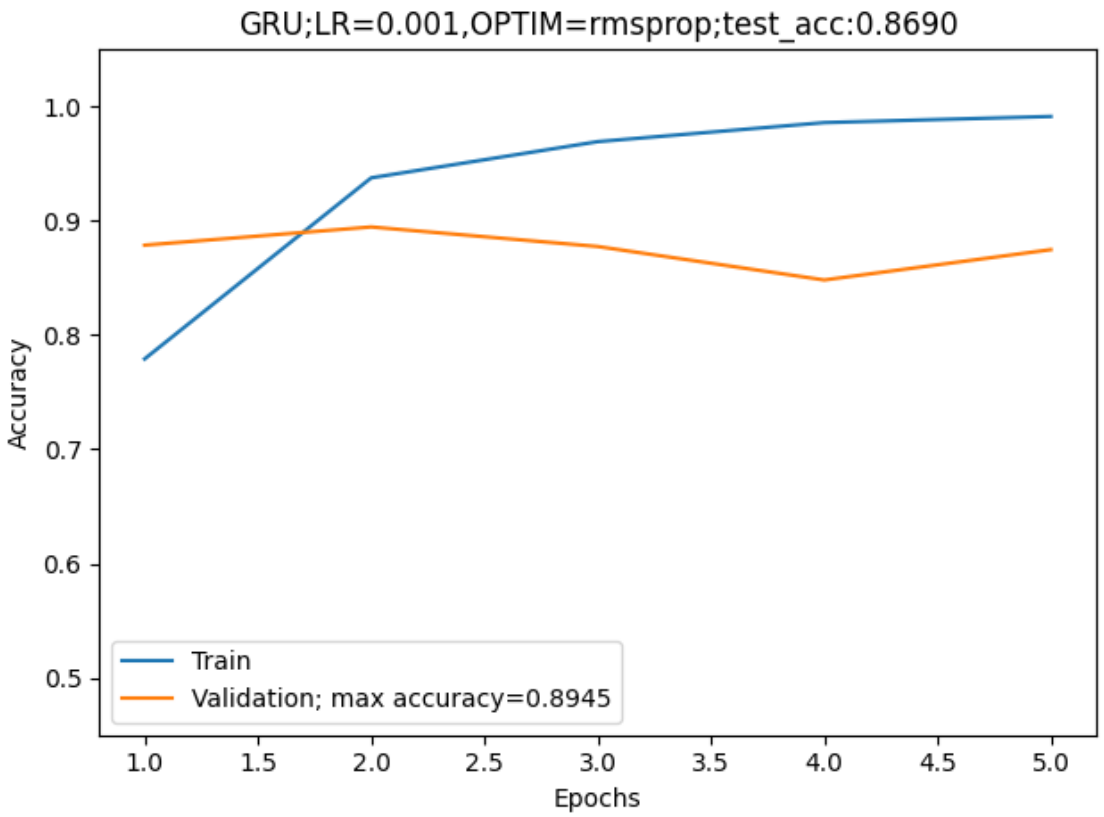


[27]:
```
q2b_rmsprop_hparams = HyperParams()
q2b_rmsprop_hparams.LR = 0.001
q2b_rmsprop_hparams.OPTIM = 'rmsprop'
rd_2b_rmsprop = train_and_test_model_with_hparams(q2b_rmsprop_hparams, "gru",
                                                  override_models_with_gru=True)
```

```
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 87,831 trainable parameters
Saving …
epoch: 1
train_loss: 0.432, train_acc: 0.779
valid_loss: 0.295, valid_acc: 0.879
Saving …
epoch: 2
train_loss: 0.171, train_acc: 0.937
valid_loss: 0.259, valid_acc: 0.894
Saving …
epoch: 3
```

```
train_loss: 0.086, train_acc: 0.969
valid_loss: 0.363, valid_acc: 0.877
Saving …
epoch: 4
train_loss: 0.041, train_acc: 0.986
valid_loss: 0.511, valid_acc: 0.848
Saving …
epoch: 5
train_loss: 0.026, train_acc: 0.991
valid_loss: 0.585, valid_acc: 0.875
test_loss: 0.606, test_acc: 0.869
```

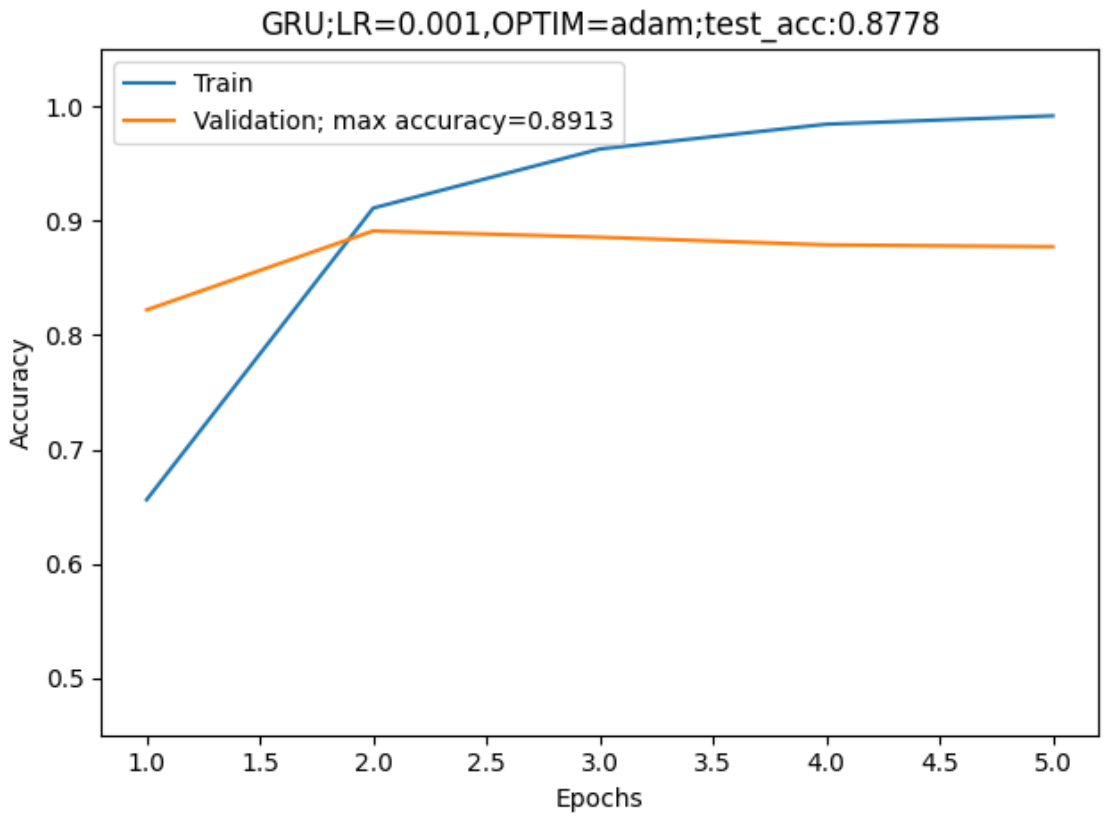
[28]:
```
plot_train_val_acc(save_name='q2b_rmsprop', ret_dict=rd_2b_rmsprop,␣
↪orig_hp=ORIG_HPARAMS,
                   new_hp=q2b_rmsprop_hparams, save='y', gru=True)
```



[29]:
```
q2b_adam_hparams = HyperParams()
q2b_adam_hparams.LR = 0.001
q2b_adam_hparams.OPTIM = 'adam'
rd_2b_adam = train_and_test_model_with_hparams(q2b_adam_hparams, "gru",
                                               override_models_with_gru=True)
```

```
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 87,831 trainable parameters
Saving …
epoch: 1
train_loss: 0.590, train_acc: 0.656
valid_loss: 0.439, valid_acc: 0.822
Saving …
epoch: 2
train_loss: 0.227, train_acc: 0.911
valid_loss: 0.260, valid_acc: 0.891
Saving …
epoch: 3
train_loss: 0.113, train_acc: 0.963
valid_loss: 0.337, valid_acc: 0.886
Saving …
epoch: 4
train_loss: 0.051, train_acc: 0.985
valid_loss: 0.389, valid_acc: 0.879
Saving …
epoch: 5
train_loss: 0.027, train_acc: 0.992
valid_loss: 0.632, valid_acc: 0.878
test_loss: 0.637, test_acc: 0.878
```

[30]:
```
plot_train_val_acc(save_name='q2b_adam', ret_dict=rd_2b_adam,␣
↪orig_hp=ORIG_HPARAMS,
                   new_hp=q2b_adam_hparams, save='y', gru=True)
```



1.6.8 Lab 2 (c) Deeper LSTMs

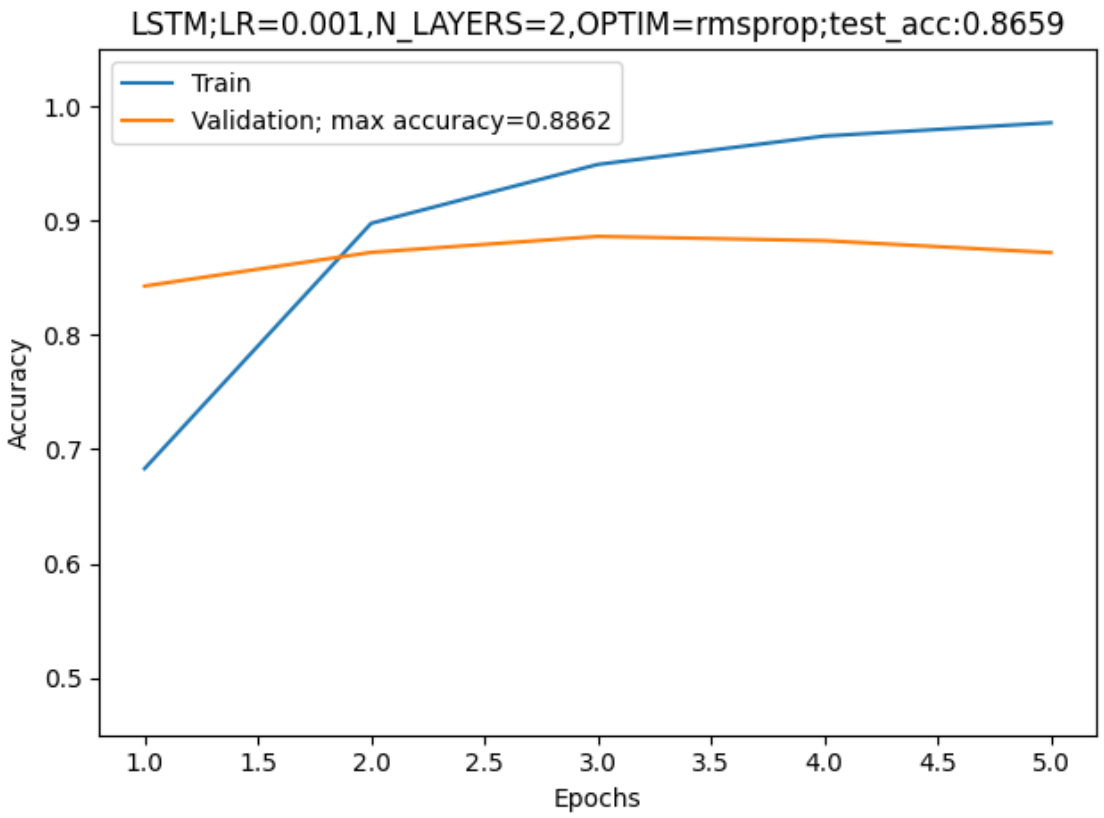
[31]:
```
# Try to make your RNN model deeper by changing the number of layers.
# Is your RNN model achieving a better accuracy on IDMB classification?
# You may use LSTM as an example.
# Hint: you do not need to explore more than 4 recurrent layers in a RNN model

def assert_n_layers(hp_):
    n_layers_min, n_layers_max = 2, 4
    assert hp_.N_LAYERS >= 2 and hp_.N_LAYERS <= 4


def train_lab2c(num_layers):
    # using lstm. returns ret_dict and hyperparameters instance, changes␣
 ↪N_LAYERS
    q2c_rmsprop_HP = HyperParams()
    q2c_rmsprop_HP.LR = 0.001
    q2c_rmsprop_HP.OPTIM = 'rmsprop'

    q2c_rmsprop_HP.N_LAYERS = num_layers
```

```
    assert_n_layers(hp_=q2c_rmsprop_HP)

    ret_dict_2c = train_and_test_model_with_hparams(hparams=q2c_rmsprop_HP,␣
↪model_type="lstm")
    return ret_dict_2c, q2c_rmsprop_HP
```

[32]:
```
q2c_rd2, q2c_hp2 = train_lab2c(num_layers=2)
```

```
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 178,931 trainable parameters
Saving …
epoch: 1
train_loss: 0.573, train_acc: 0.683
valid_loss: 0.370, valid_acc: 0.843
Saving …
epoch: 2
train_loss: 0.256, train_acc: 0.898
valid_loss: 0.294, valid_acc: 0.872
Saving …
epoch: 3
train_loss: 0.141, train_acc: 0.949
valid_loss: 0.308, valid_acc: 0.886
Saving …
epoch: 4
train_loss: 0.079, train_acc: 0.974
valid_loss: 0.315, valid_acc: 0.882
Saving …
epoch: 5
train_loss: 0.046, train_acc: 0.986
valid_loss: 0.385, valid_acc: 0.872
test_loss: 0.396, test_acc: 0.866
```

[33]:
```
plot_train_val_acc(save_name='q2c', ret_dict=q2c_rd2, orig_hp=ORIG_HPARAMS,
                   new_hp=q2c_hp2, save='y', gru=False)
```

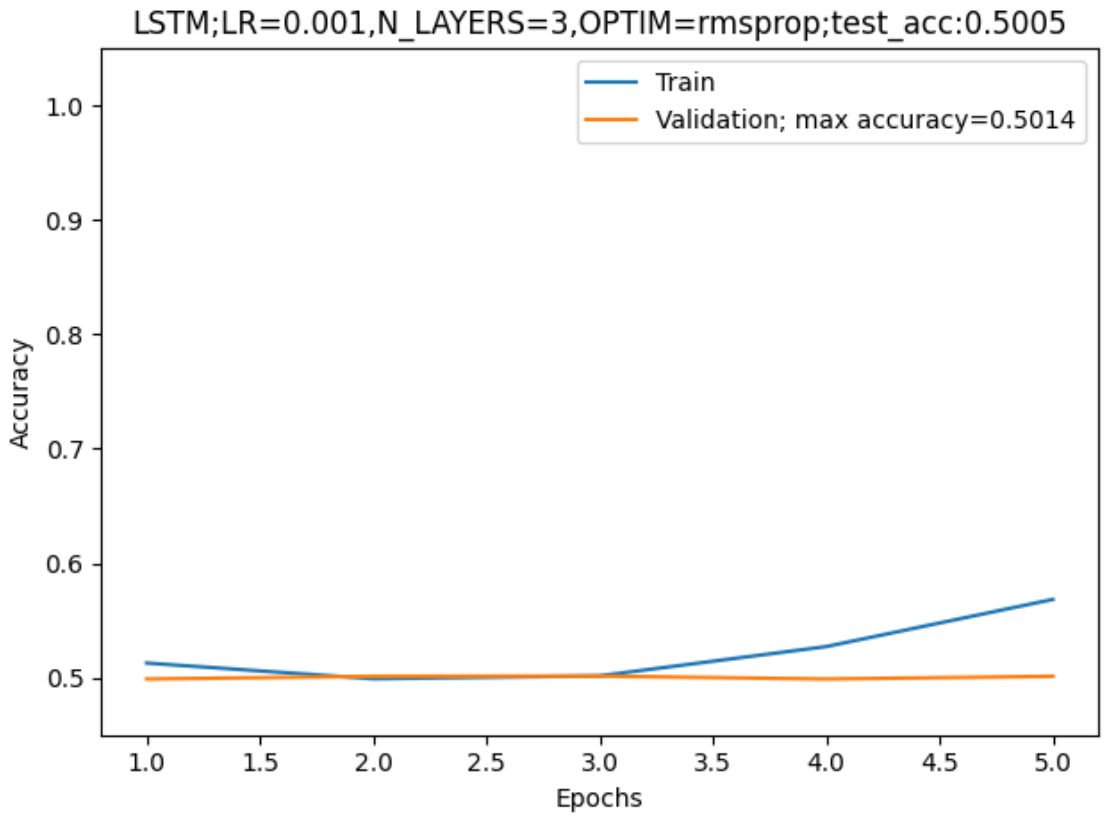


[34]:
```
q2c_rd3, q2c_hp3 = train_lab2c(num_layers=3)
```

```
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 259,731 trainable parameters
Saving …
epoch: 1
train_loss: 0.691, train_acc: 0.513
valid_loss: 0.693, valid_acc: 0.499
Saving …
epoch: 2
train_loss: 0.694, train_acc: 0.499
valid_loss: 0.693, valid_acc: 0.501
Saving …
epoch: 3
train_loss: 0.693, train_acc: 0.502
valid_loss: 0.697, valid_acc: 0.501
Saving …
epoch: 4
```

```
train_loss: 0.692, train_acc: 0.527
valid_loss: 0.699, valid_acc: 0.499
Saving …
epoch: 5
train_loss: 0.683, train_acc: 0.568
valid_loss: 0.685, valid_acc: 0.501
test_loss: 0.687, test_acc: 0.500
```

[35]:
```
plot_train_val_acc(save_name='q2c', ret_dict=q2c_rd3, orig_hp=ORIG_HPARAMS,
                   new_hp=q2c_hp3, save='y', gru=False)
```

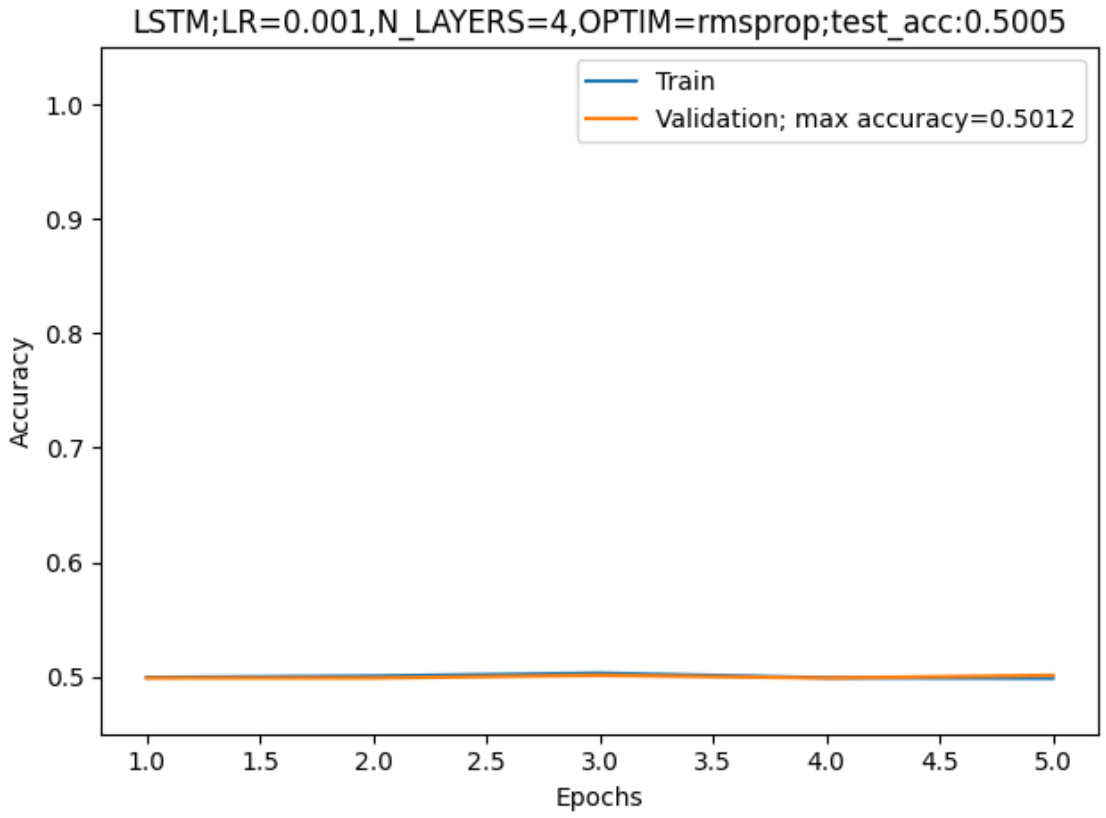


[36]:
```
q2c_rd4, q2c_hp4 = train_lab2c(num_layers=4)
```

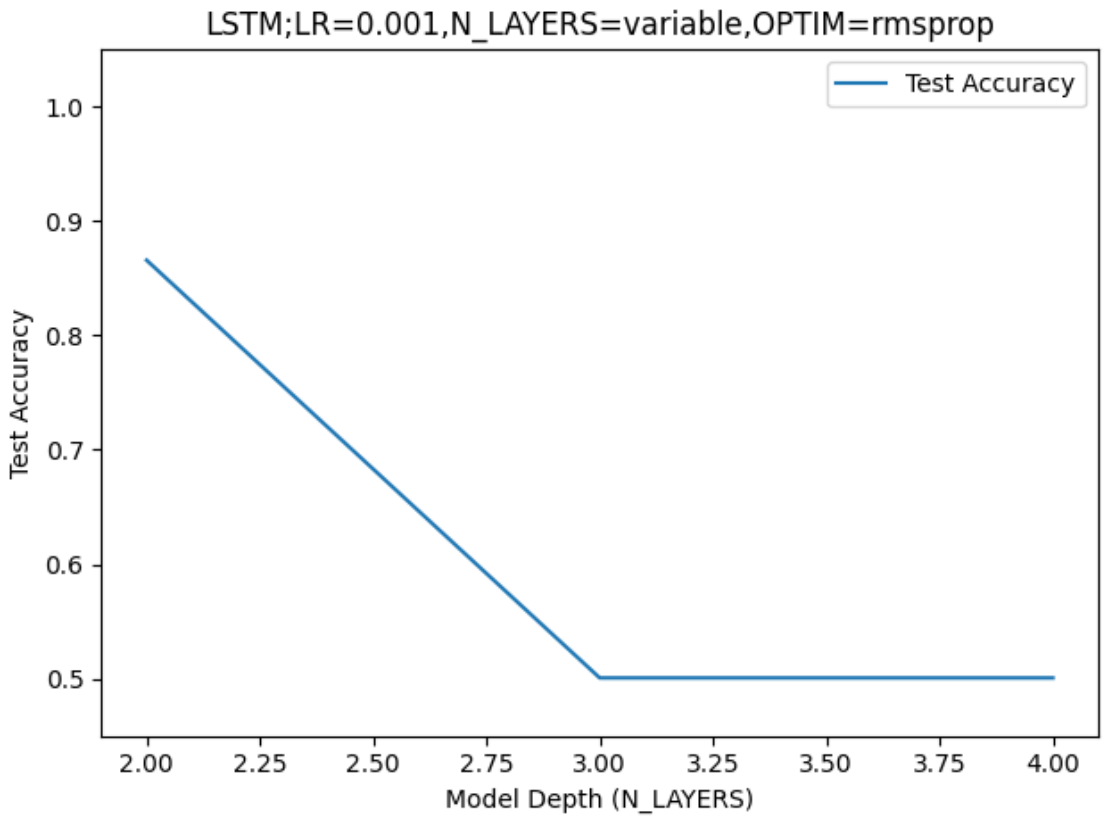
```
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 340,531 trainable parameters
Saving …
epoch: 1
train_loss: 0.693, train_acc: 0.499
```

```
valid_loss: 0.693, valid_acc: 0.499
Saving …
epoch: 2
train_loss: 0.694, train_acc: 0.500
valid_loss: 0.693, valid_acc: 0.499
Saving …
epoch: 3
train_loss: 0.693, train_acc: 0.503
valid_loss: 0.693, valid_acc: 0.501
Saving …
epoch: 4
train_loss: 0.694, train_acc: 0.499
valid_loss: 0.693, valid_acc: 0.499
Saving …
epoch: 5
train_loss: 0.693, train_acc: 0.499
valid_loss: 0.693, valid_acc: 0.501
test_loss: 0.693, test_acc: 0.500
```

[37]:
```
plot_train_val_acc(save_name='q2c', ret_dict=q2c_rd4, orig_hp=ORIG_HPARAMS,
                   new_hp=q2c_hp4, save='y', gru=False)
```



[78]:
```
test_losses_q2c = [q2c_rd2['test_acc'], q2c_rd3['test_acc'],␣
 ↪q2c_rd4['test_acc']]
q2c_nlayers = [2, 3, 4]

fig, ax = plt.subplots(1, 1)

ax.plot(q2c_nlayers, test_losses_q2c, label='Test Accuracy')

ax.set_ylim([0.45, 1.05])

# diff_param = diff_hparams(orig_hp=orig_hp, new_hp=new_hp)

title_ = "LSTM;LR=0.001,N_LAYERS=variable,OPTIM=rmsprop"

ax.set_xlabel('Model Depth (N_LAYERS)')
ax.set_ylabel('Test Accuracy')
ax.legend()
ax.set_title(title_)
fig.tight_layout()

plt.savefig('q2c_overall.pdf', dpi=500, bbox_inches='tight')
```



1.6.9 Lab 2 (d) Wider LSTMs

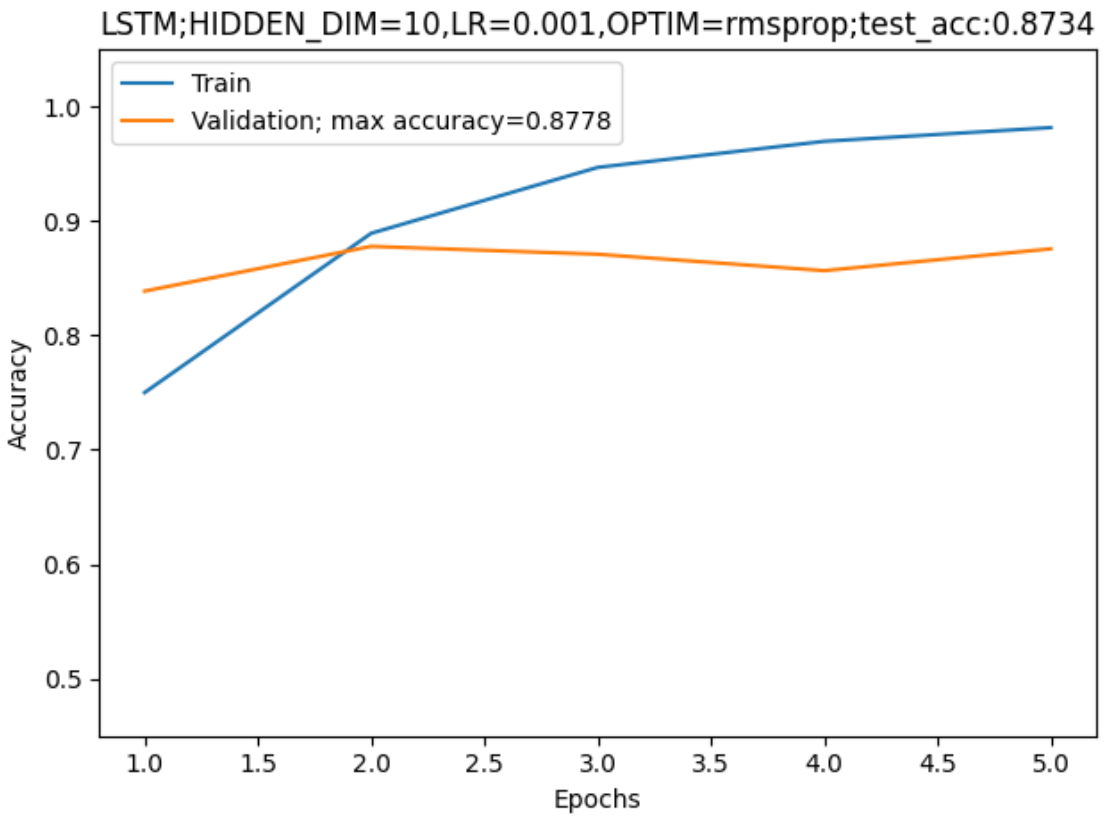
[38]:
```
# Try to make your RNN model wider by changing the number of hidden units.
# Is your RNN model achieving a better accuracy on IDMB classification? You may␣
 ↪use LSTM as an example.
# (Hint: you do not need to explore a hidden dimension of more than 320 on␣
 ↪IMDB).

def assert_h_dim(hp_):
    hidden_dim_max = 320
    assert hp_.HIDDEN_DIM <= 320 and hp_.HIDDEN_DIM >=10


def train_lab2d(hidden_dimension):
    # HIDDEN_DIM
    # using lstm. returns ret_dict and hyperparameters instance, changes␣
 ↪HIDDEN_DIM
    q2d_rmsprop_HP = HyperParams()
    q2d_rmsprop_HP.LR = 0.001
    q2d_rmsprop_HP.OPTIM = 'rmsprop'

    q2d_rmsprop_HP.HIDDEN_DIM = hidden_dimension
    assert_h_dim(hp_=q2d_rmsprop_HP)

    ret_dict_2d = train_and_test_model_with_hparams(hparams=q2d_rmsprop_HP,␣
 ↪model_type="lstm")
    return ret_dict_2d, q2d_rmsprop_HP
```

[39]:
```
q2d_rd_10, q2d_hp_10 = train_lab2d(hidden_dimension=10)
```

```
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 57,271 trainable parameters
Saving …
epoch: 1
train_loss: 0.506, train_acc: 0.750
valid_loss: 0.395, valid_acc: 0.839
Saving …
epoch: 2
train_loss: 0.287, train_acc: 0.889
valid_loss: 0.330, valid_acc: 0.878
Saving …
epoch: 3
train_loss: 0.152, train_acc: 0.947
valid_loss: 0.330, valid_acc: 0.871
```

```
Saving …
epoch: 4
train_loss: 0.093, train_acc: 0.970
valid_loss: 0.378, valid_acc: 0.857
Saving …
epoch: 5
train_loss: 0.060, train_acc: 0.982
valid_loss: 0.430, valid_acc: 0.876
test_loss: 0.457, test_acc: 0.873
```

[40]:
```
plot_train_val_acc(save_name='q2d', ret_dict=q2d_rd_10, orig_hp=ORIG_HPARAMS,
                   new_hp=q2d_hp_10, save='y', gru=False)
```

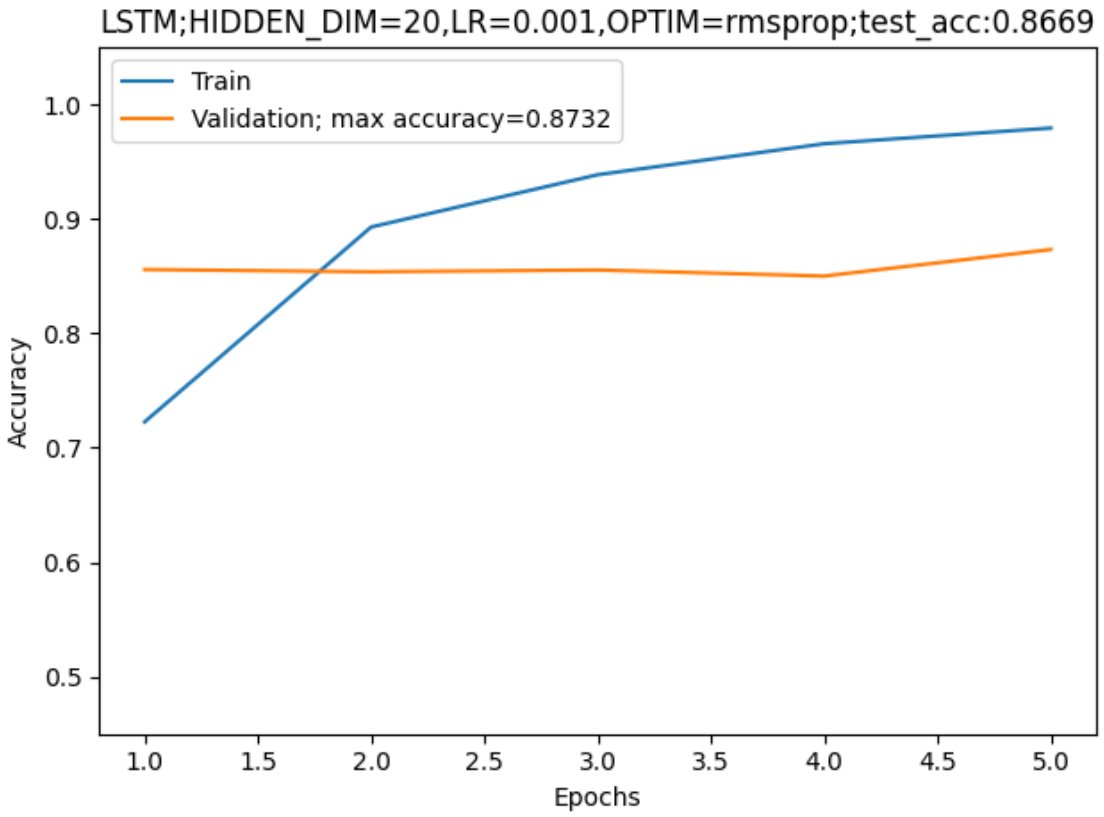


[41]:
```
q2d_rd_20, q2d_hp_20 = train_lab2d(hidden_dimension=20)
```

```
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 58,611 trainable parameters
Saving …
```

```
epoch: 1
train_loss: 0.543, train_acc: 0.723
valid_loss: 0.367, valid_acc: 0.856
Saving …
epoch: 2
train_loss: 0.282, train_acc: 0.893
valid_loss: 0.348, valid_acc: 0.854
Saving …
epoch: 3
train_loss: 0.175, train_acc: 0.939
valid_loss: 0.431, valid_acc: 0.855
Saving …
epoch: 4
train_loss: 0.103, train_acc: 0.966
valid_loss: 0.395, valid_acc: 0.850
Saving …
epoch: 5
train_loss: 0.066, train_acc: 0.979
valid_loss: 0.449, valid_acc: 0.873
test_loss: 0.477, test_acc: 0.867
```

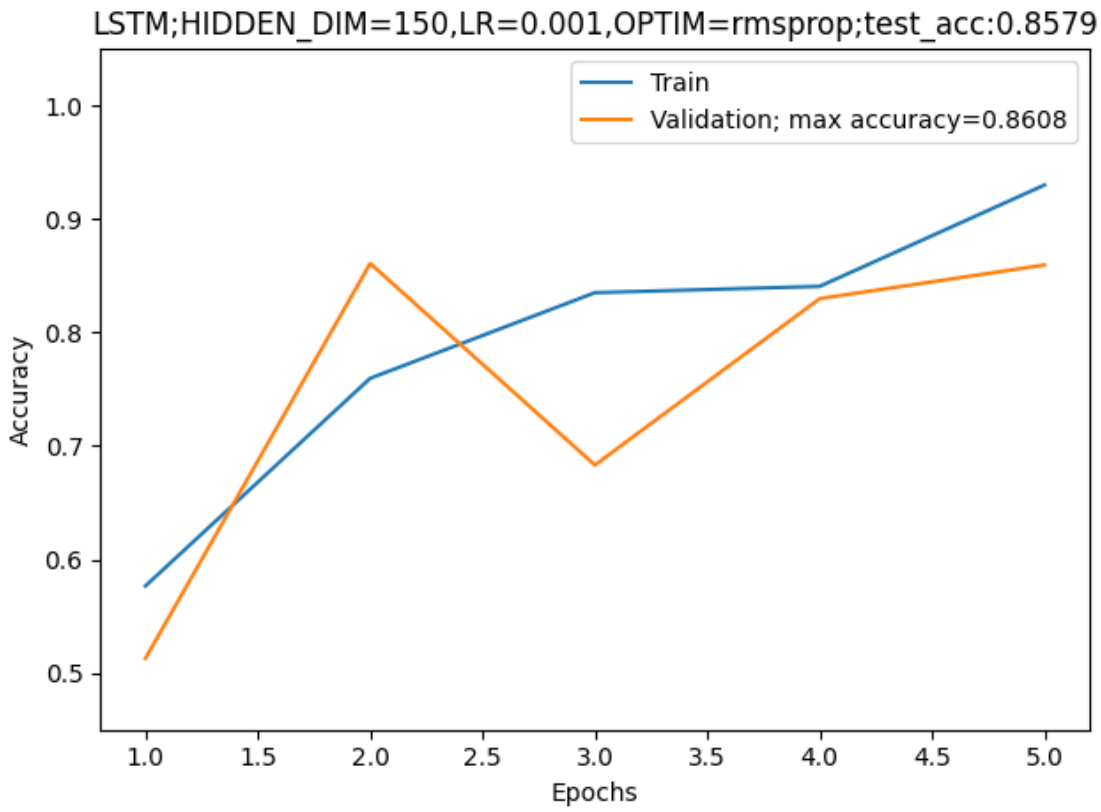
[42]:
```
plot_train_val_acc(save_name='q2d', ret_dict=q2d_rd_20, orig_hp=ORIG_HPARAMS,
                   new_hp=q2d_hp_20, save='y', gru=False)
```



[43]:
```
q2d_rd_150, q2d_hp_150 = train_lab2d(hidden_dimension=150)
```

```
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 148,831 trainable parameters
Saving …
epoch: 1
train_loss: 0.692, train_acc: 0.577
valid_loss: 0.766, valid_acc: 0.513
Saving …
epoch: 2
train_loss: 0.496, train_acc: 0.760
valid_loss: 0.353, valid_acc: 0.861
Saving …
epoch: 3
train_loss: 0.384, train_acc: 0.835
valid_loss: 0.592, valid_acc: 0.683
Saving …
epoch: 4
train_loss: 0.361, train_acc: 0.841
valid_loss: 0.413, valid_acc: 0.830
Saving …
epoch: 5
train_loss: 0.186, train_acc: 0.930
valid_loss: 0.356, valid_acc: 0.859
test_loss: 0.370, test_acc: 0.858
```

[44]:
```
plot_train_val_acc(save_name='q2d', ret_dict=q2d_rd_150, orig_hp=ORIG_HPARAMS,
                   new_hp=q2d_hp_150, save='y', gru=False)
```

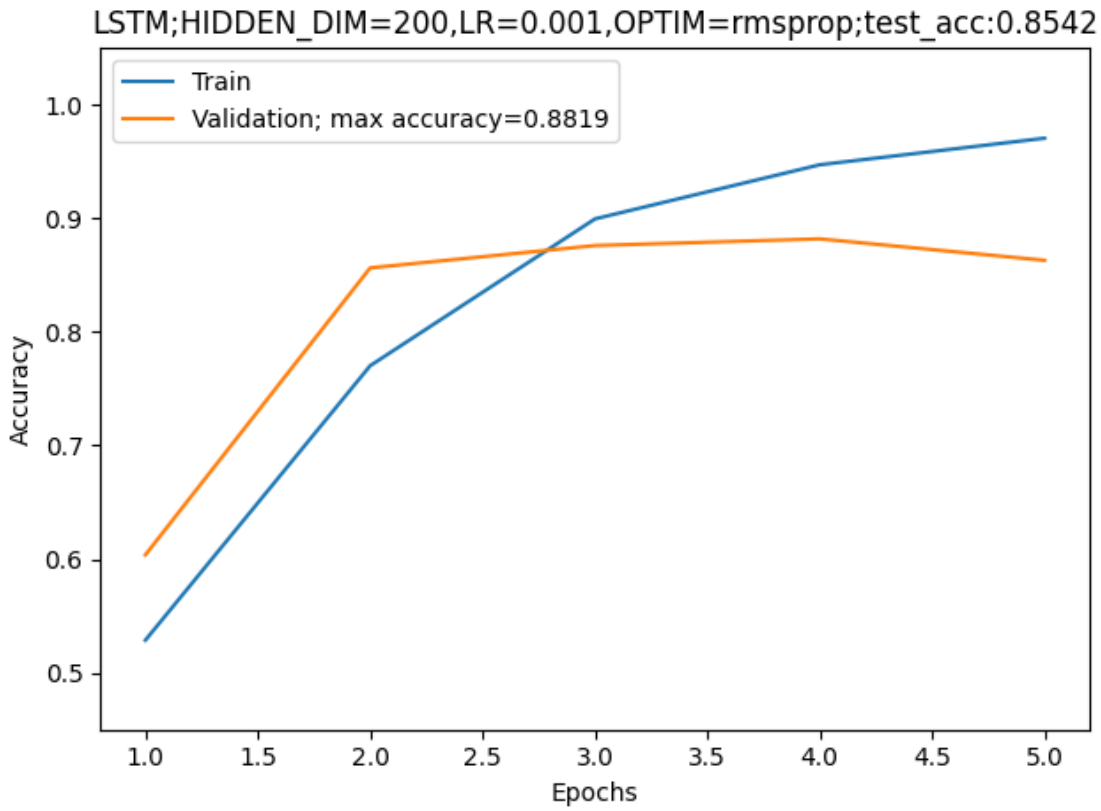


[45]:
```
q2d_rd_200, q2d_hp_200 = train_lab2d(hidden_dimension=200)
```

```
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 219,531 trainable parameters
Saving …
epoch: 1
train_loss: 0.784, train_acc: 0.529
valid_loss: 0.649, valid_acc: 0.604
Saving …
epoch: 2
train_loss: 0.483, train_acc: 0.770
valid_loss: 0.366, valid_acc: 0.856
Saving …
epoch: 3
train_loss: 0.262, train_acc: 0.899
valid_loss: 0.302, valid_acc: 0.876
Saving …
epoch: 4
```

```
train_loss: 0.150, train_acc: 0.947
valid_loss: 0.319, valid_acc: 0.882
Saving …
epoch: 5
train_loss: 0.088, train_acc: 0.970
valid_loss: 0.401, valid_acc: 0.863
test_loss: 0.420, test_acc: 0.854
```

[46]:
```
plot_train_val_acc(save_name='q2d', ret_dict=q2d_rd_200, orig_hp=ORIG_HPARAMS,
                   new_hp=q2d_hp_200, save='y', gru=False)
```

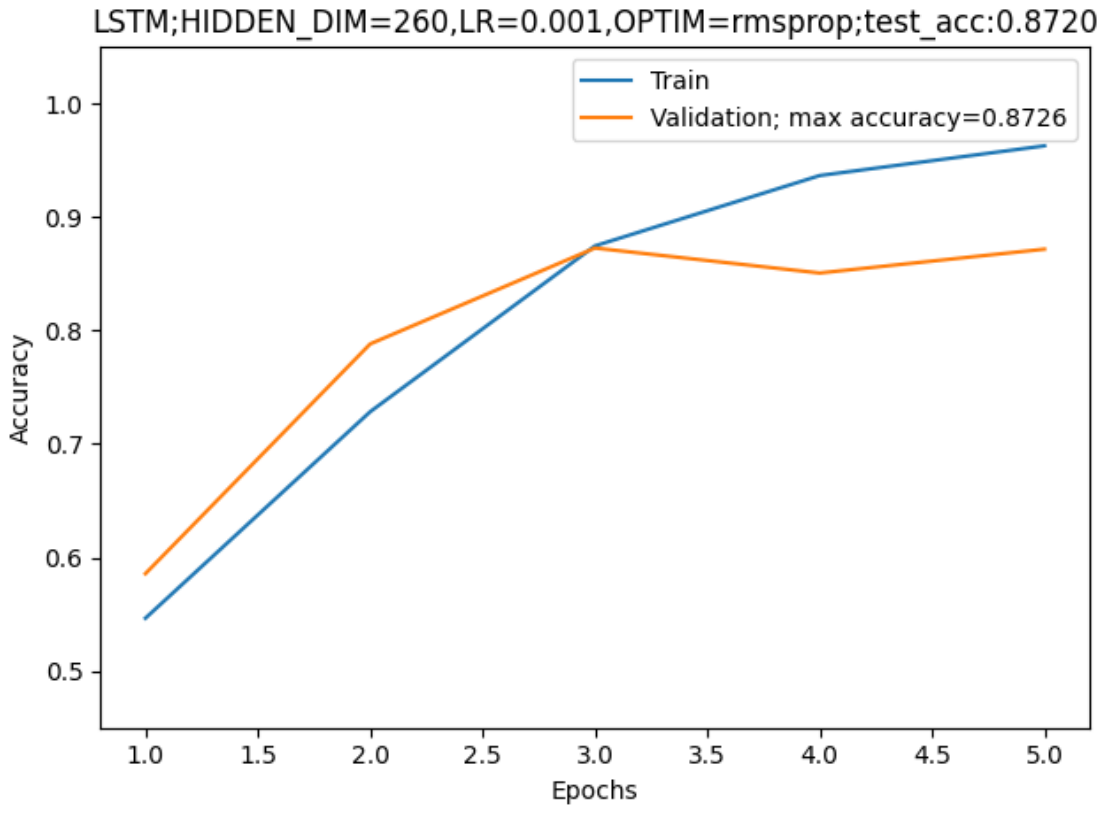


[47]:
```
q2d_rd_260, q2d_hp_260 = train_lab2d(hidden_dimension=260)
```

```
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 330,771 trainable parameters
Saving …
epoch: 1
train_loss: 0.711, train_acc: 0.546
valid_loss: 0.700, valid_acc: 0.586
```

```
Saving …
epoch: 2
train_loss: 0.543, train_acc: 0.728
valid_loss: 0.467, valid_acc: 0.788
Saving …
epoch: 3
train_loss: 0.311, train_acc: 0.875
valid_loss: 0.332, valid_acc: 0.873
Saving …
epoch: 4
train_loss: 0.169, train_acc: 0.937
valid_loss: 0.375, valid_acc: 0.851
Saving …
epoch: 5
train_loss: 0.108, train_acc: 0.963
valid_loss: 0.345, valid_acc: 0.872
test_loss: 0.359, test_acc: 0.872
```

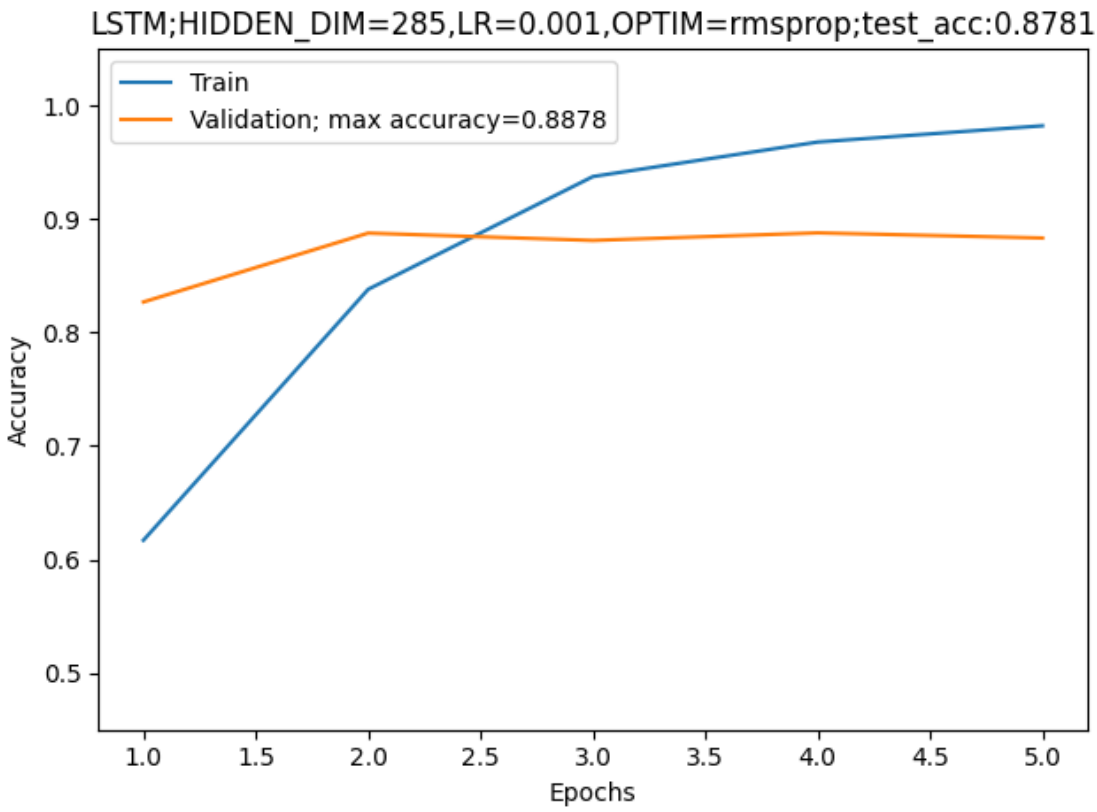
[48]:
```
plot_train_val_acc(save_name='q2d', ret_dict=q2d_rd_260, orig_hp=ORIG_HPARAMS,
                   new_hp=q2d_hp_260, save='y', gru=False)
```



[55]:
```
q2d_rd_285, q2d_hp_285 = train_lab2d(hidden_dimension=285)
```

```
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 385,621 trainable parameters
Saving …
epoch: 1
train_loss: 0.669, train_acc: 0.617
valid_loss: 0.398, valid_acc: 0.827
Saving …
epoch: 2
train_loss: 0.368, train_acc: 0.838
valid_loss: 0.284, valid_acc: 0.888
Saving …
epoch: 3
train_loss: 0.169, train_acc: 0.937
valid_loss: 0.302, valid_acc: 0.881
Saving …
epoch: 4
train_loss: 0.093, train_acc: 0.968
valid_loss: 0.315, valid_acc: 0.888
Saving …
epoch: 5
train_loss: 0.055, train_acc: 0.982
valid_loss: 0.443, valid_acc: 0.883
test_loss: 0.473, test_acc: 0.878
```

[56]:
```
plot_train_val_acc(save_name='q2d', ret_dict=q2d_rd_285, orig_hp=ORIG_HPARAMS,
                   new_hp=q2d_hp_285, save='y', gru=False)
```

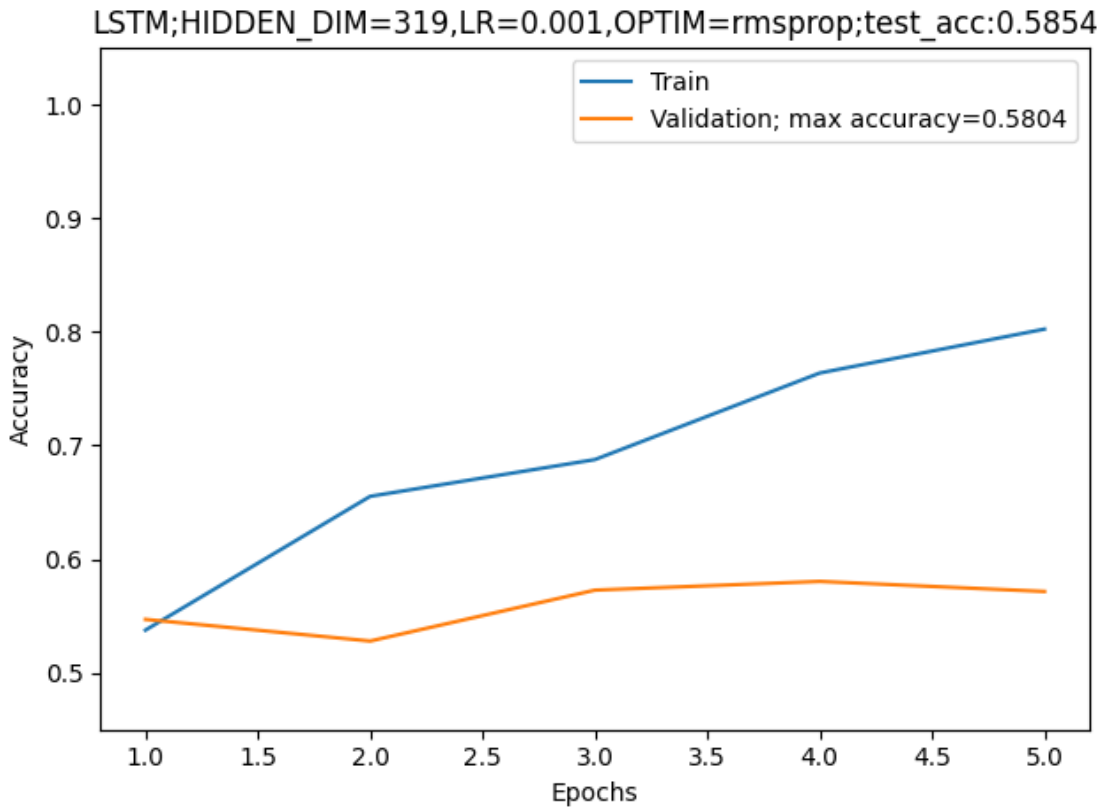


[52]:
```
q2d_rd_319, q2d_hp_319 = train_lab2d(hidden_dimension=319)
```

```
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 468,241 trainable parameters
Saving …
epoch: 1
train_loss: 0.750, train_acc: 0.538
valid_loss: 0.686, valid_acc: 0.547
Saving …
epoch: 2
train_loss: 0.618, train_acc: 0.655
valid_loss: 1.171, valid_acc: 0.528
Saving …
epoch: 3
train_loss: 0.590, train_acc: 0.688
valid_loss: 0.747, valid_acc: 0.573
Saving …
epoch: 4
```

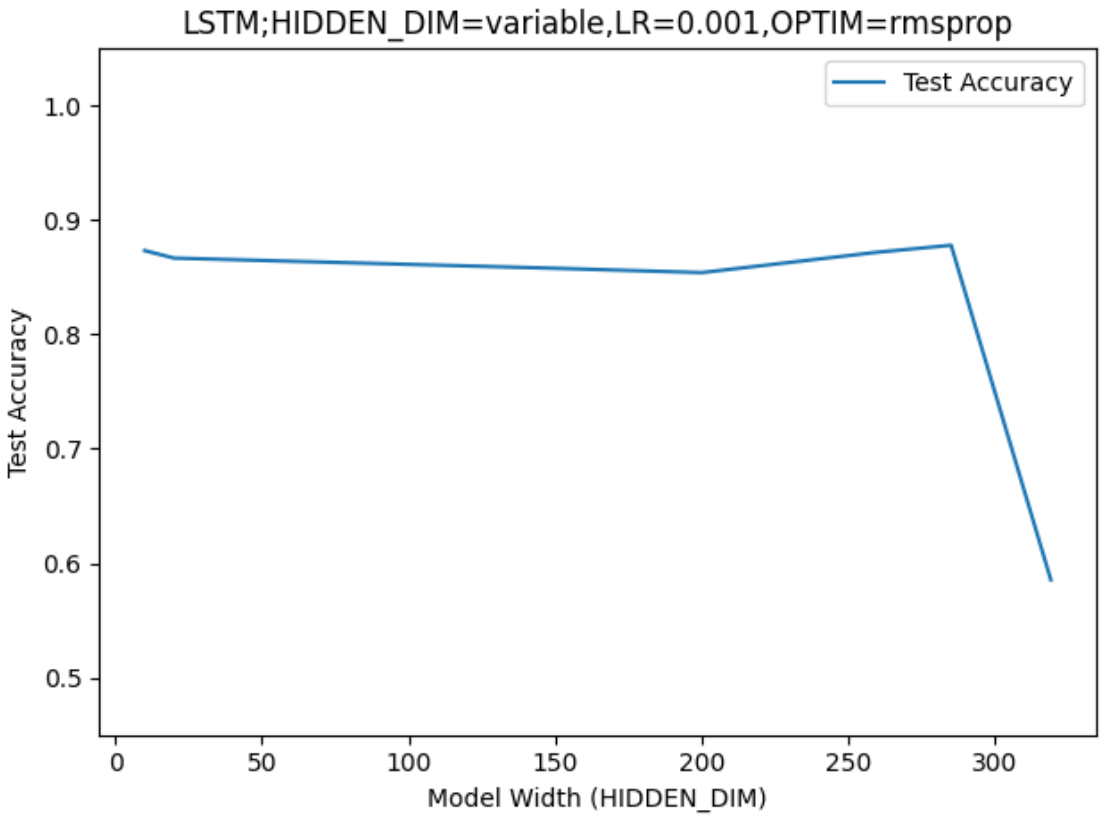
```
train_loss: 0.491, train_acc: 0.764
valid_loss: 0.762, valid_acc: 0.580
Saving …
epoch: 5
train_loss: 0.433, train_acc: 0.802
valid_loss: 0.869, valid_acc: 0.572
test_loss: 0.843, test_acc: 0.585
```

[54]:
```
plot_train_val_acc(save_name='q2d', ret_dict=q2d_rd_319, orig_hp=ORIG_HPARAMS,
                   new_hp=q2d_hp_319, save='y', gru=False)
```



[82]:
```
test_losses_q2d = [q2d_rd_10['test_acc'], q2d_rd_20['test_acc'],␣
 ↪q2d_rd_150['test_acc'],
                   q2d_rd_200['test_acc'], q2d_rd_260['test_acc'],␣
 ↪q2d_rd_285['test_acc'],
                   q2d_rd_319['test_acc']]
q2d_hidden_dim = [10, 20, 150, 200, 260, 285, 319]

fig, ax = plt.subplots(1, 1)

ax.plot(q2d_hidden_dim, test_losses_q2d, label='Test Accuracy')
```

```
ax.set_ylim([0.45, 1.05])

# diff_param = diff_hparams(orig_hp=orig_hp, new_hp=new_hp)

title_ = "LSTM;HIDDEN_DIM=variable,LR=0.001,OPTIM=rmsprop"

ax.set_xlabel('Model Width (HIDDEN_DIM)')
ax.set_ylabel('Test Accuracy')
ax.legend()
ax.set_title(title_)
fig.tight_layout()
plt.savefig('q2d_overall.pdf', dpi=500, bbox_inches='tight')
```



1.6.10 Lab 2 (e) Larger Embedding Table

[57]:
```
# (e) (5 pts) Embedding tables contain rich information of the input words and␣
 ↪help build a more
# powerful representation with word vectors. Try to increase the dimension of␣
 ↪embeddings.
```

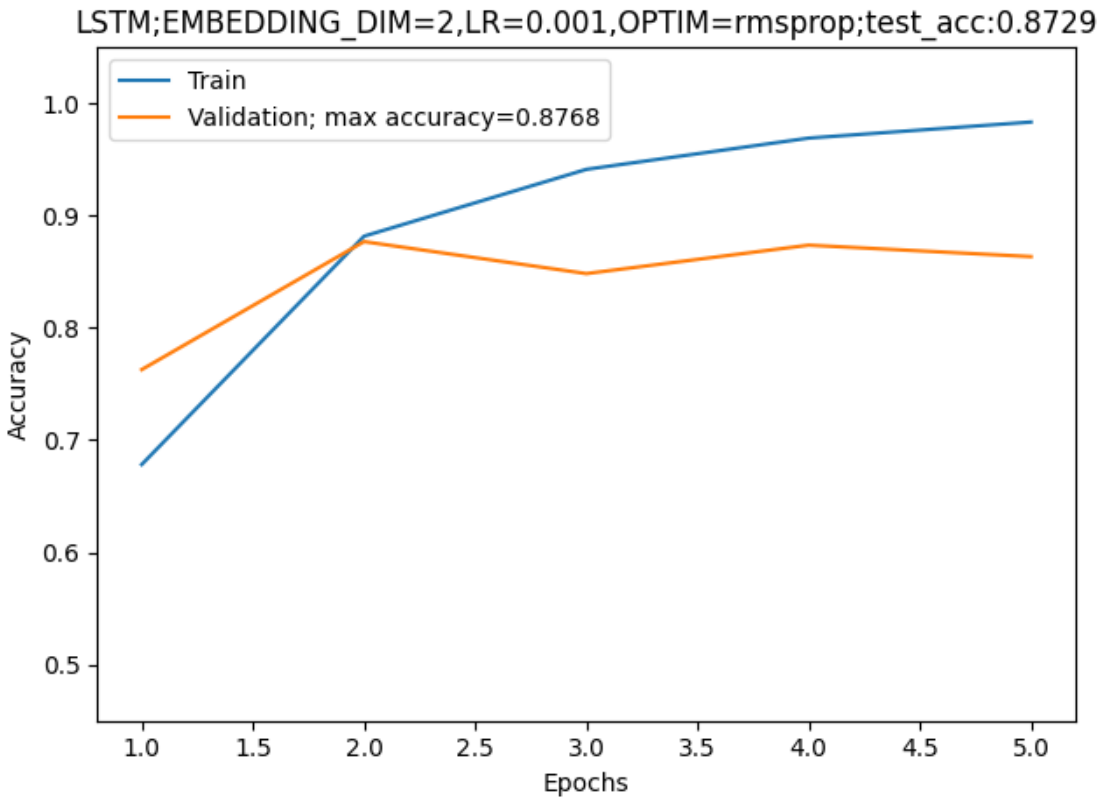
```
# Is your RNN model achieving a better accuracy on IMDB classification?
# You may use LSTM as an example.
# (Hint: you do not need to explore an embedding dimension of larger than 256).

def assert_embedding_dim(hp_):
    embedding_dim_max = 256
    assert hp_.EMBEDDING_DIM <= embedding_dim_max and hp_.EMBEDDING_DIM >= 1


def train_lab2e(embedding_dimension):
    # HIDDEN_DIM
    # using lstm. returns ret_dict and hyperparameters instance, changes␣
 ↪HIDDEN_DIM
    q2e_rmsprop_HP = HyperParams()
    q2e_rmsprop_HP.LR = 0.001
    q2e_rmsprop_HP.OPTIM = 'rmsprop'

    q2e_rmsprop_HP.EMBEDDING_DIM = embedding_dimension
    assert_h_dim(hp_=q2e_rmsprop_HP)

    ret_dict_2e = train_and_test_model_with_hparams(hparams=q2e_rmsprop_HP,␣
 ↪model_type="lstm")
    return ret_dict_2e, q2e_rmsprop_HP
```

[58]:
```
q2e_rd_2, q2e_hp_2 = train_lab2e(embedding_dimension=2)
```

```
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 155,260 trainable parameters
Saving …
epoch: 1
train_loss: 0.627, train_acc: 0.678
valid_loss: 0.515, valid_acc: 0.763
Saving …
epoch: 2
train_loss: 0.299, train_acc: 0.882
valid_loss: 0.320, valid_acc: 0.877
Saving …
epoch: 3
train_loss: 0.167, train_acc: 0.941
valid_loss: 0.346, valid_acc: 0.848
Saving …
epoch: 4
train_loss: 0.093, train_acc: 0.969
valid_loss: 0.368, valid_acc: 0.874
```

```
Saving …
epoch: 5
train_loss: 0.054, train_acc: 0.983
valid_loss: 0.445, valid_acc: 0.863
test_loss: 0.439, test_acc: 0.873
```

[59]:
```
plot_train_val_acc(save_name='q2e', ret_dict=q2e_rd_2, orig_hp=ORIG_HPARAMS,
                   new_hp=q2e_hp_2, save='y', gru=False)
```

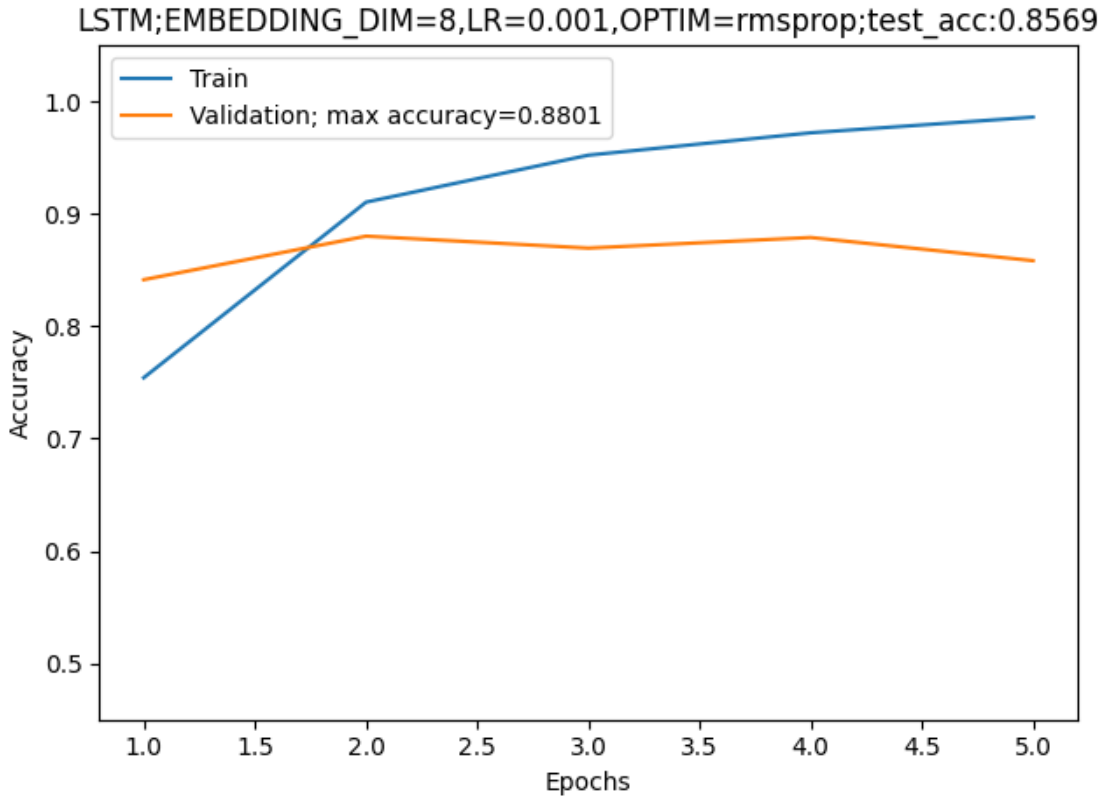


[60]:
```
q2e_rd_8, q2e_hp_8 = train_lab2e(embedding_dimension=8)
```

```
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 498,034 trainable parameters
Saving …
epoch: 1
train_loss: 0.491, train_acc: 0.754
valid_loss: 0.384, valid_acc: 0.841
Saving …
epoch: 2
```

```
train_loss: 0.240, train_acc: 0.910
valid_loss: 0.308, valid_acc: 0.880
Saving …
epoch: 3
train_loss: 0.138, train_acc: 0.952
valid_loss: 0.330, valid_acc: 0.869
Saving …
epoch: 4
train_loss: 0.085, train_acc: 0.972
valid_loss: 0.372, valid_acc: 0.879
Saving …
epoch: 5
train_loss: 0.046, train_acc: 0.986
valid_loss: 0.440, valid_acc: 0.858
test_loss: 0.454, test_acc: 0.857
```

[61]:
```
plot_train_val_acc(save_name='q2e', ret_dict=q2e_rd_8, orig_hp=ORIG_HPARAMS,
                   new_hp=q2e_hp_8, save='y', gru=False)
```

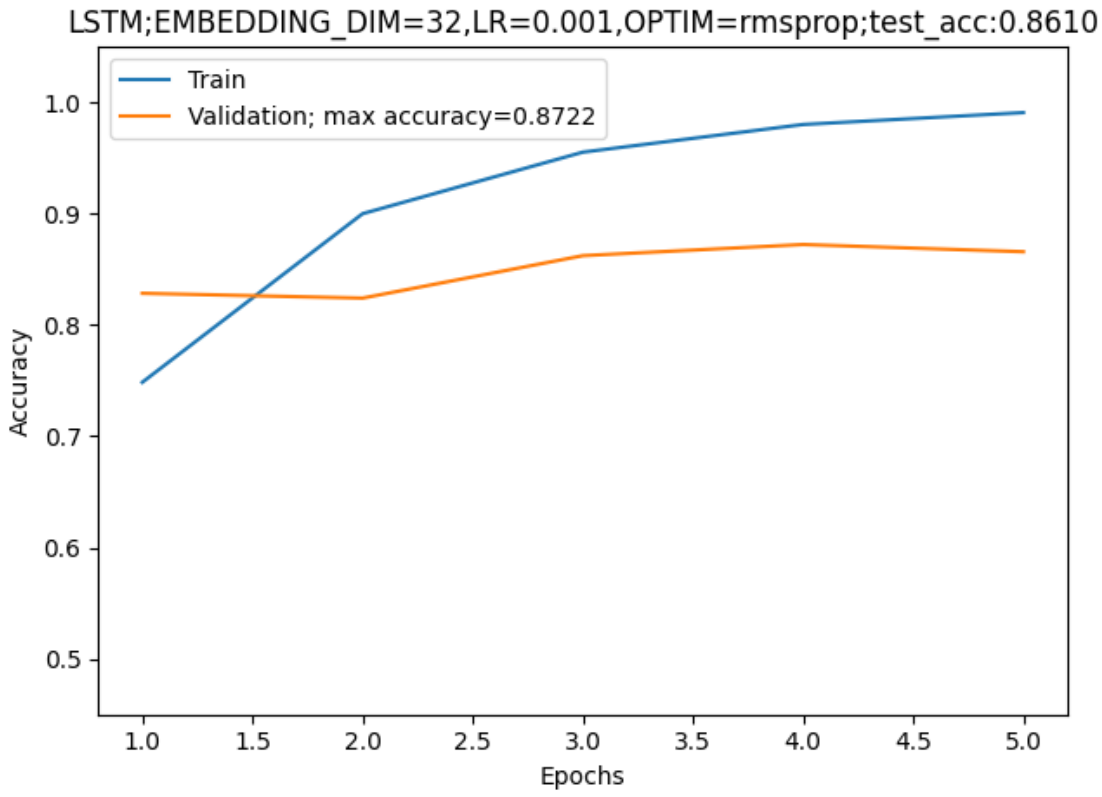


[62]:
```
q2e_rd_32, q2e_hp_32 = train_lab2e(embedding_dimension=32)
```

```
shape of train data is 35000
```

```
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 1,869,130 trainable parameters
Saving …
epoch: 1
train_loss: 0.525, train_acc: 0.748
valid_loss: 0.391, valid_acc: 0.828
Saving …
epoch: 2
train_loss: 0.258, train_acc: 0.900
valid_loss: 0.399, valid_acc: 0.824
Saving …
epoch: 3
train_loss: 0.128, train_acc: 0.955
valid_loss: 0.380, valid_acc: 0.862
Saving …
epoch: 4
train_loss: 0.062, train_acc: 0.980
valid_loss: 0.460, valid_acc: 0.872
Saving …
epoch: 5
train_loss: 0.029, train_acc: 0.991
valid_loss: 0.516, valid_acc: 0.866
test_loss: 0.565, test_acc: 0.861
```

[63]:
```
plot_train_val_acc(save_name='q2e', ret_dict=q2e_rd_32, orig_hp=ORIG_HPARAMS,
                   new_hp=q2e_hp_32, save='y', gru=False)
```

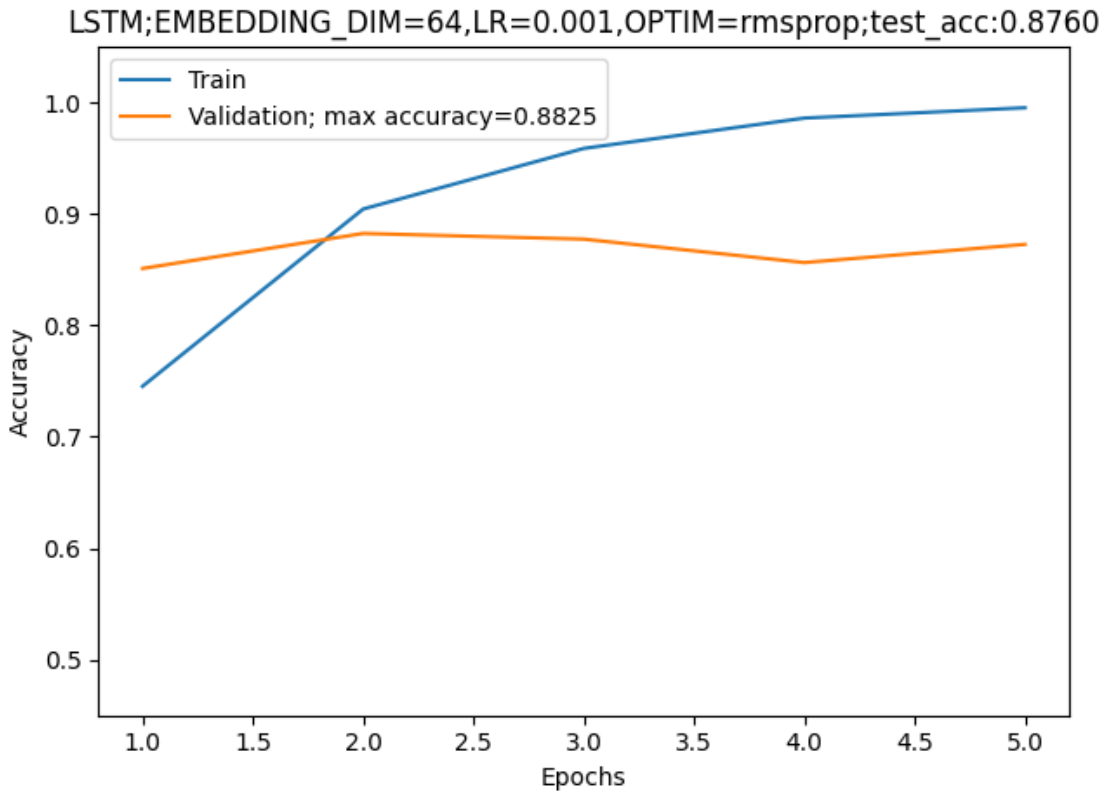


[64]:
```
q2e_rd_64, q2e_hp_64 = train_lab2e(embedding_dimension=64)
```

```
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 3,697,258 trainable parameters
Saving …
epoch: 1
train_loss: 0.515, train_acc: 0.745
valid_loss: 0.366, valid_acc: 0.851
Saving …
epoch: 2
train_loss: 0.242, train_acc: 0.904
valid_loss: 0.293, valid_acc: 0.882
Saving …
epoch: 3
train_loss: 0.114, train_acc: 0.959
valid_loss: 0.312, valid_acc: 0.877
Saving …
epoch: 4
train_loss: 0.042, train_acc: 0.986
```

```
valid_loss: 0.454, valid_acc: 0.856
Saving …
epoch: 5
train_loss: 0.016, train_acc: 0.995
valid_loss: 0.727, valid_acc: 0.873
test_loss: 0.723, test_acc: 0.876
```

[65]:
```
plot_train_val_acc(save_name='q2e', ret_dict=q2e_rd_64, orig_hp=ORIG_HPARAMS,
                   new_hp=q2e_hp_64, save='y', gru=False)
```

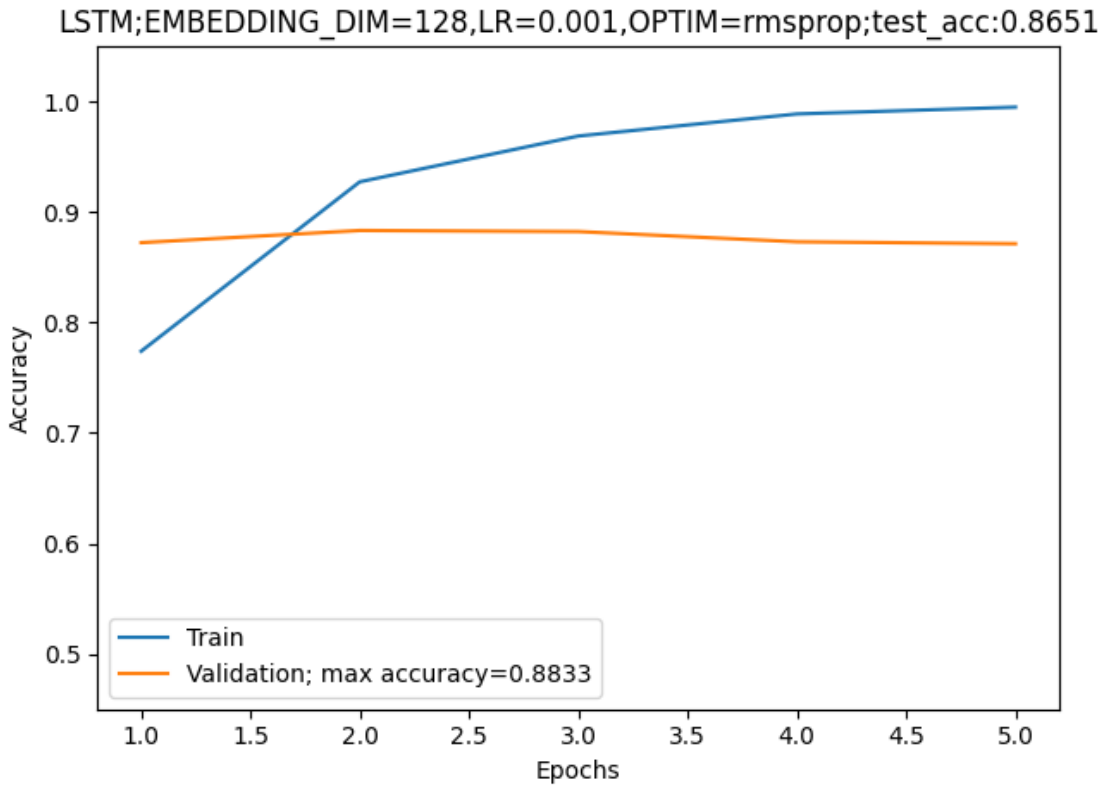


[66]:
```
q2e_rd_128, q2e_hp_128 = train_lab2e(embedding_dimension=128)
```

```
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 7,353,514 trainable parameters
Saving …
epoch: 1
train_loss: 0.468, train_acc: 0.774
valid_loss: 0.297, valid_acc: 0.872
Saving …
```

```
epoch: 2
train_loss: 0.195, train_acc: 0.927
valid_loss: 0.320, valid_acc: 0.883
Saving …
epoch: 3
train_loss: 0.090, train_acc: 0.969
valid_loss: 0.335, valid_acc: 0.882
Saving …
epoch: 4
train_loss: 0.036, train_acc: 0.989
valid_loss: 0.437, valid_acc: 0.873
Saving …
epoch: 5
train_loss: 0.016, train_acc: 0.995
valid_loss: 0.601, valid_acc: 0.871
test_loss: 0.626, test_acc: 0.865
```

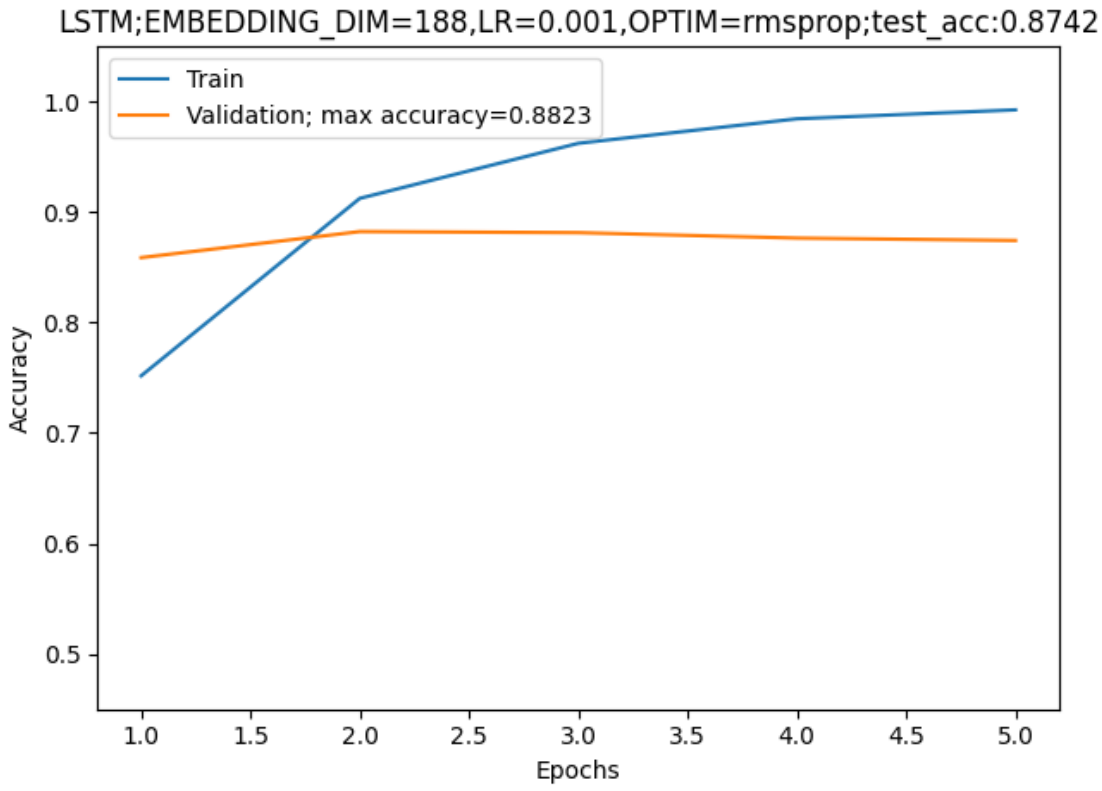
[67]:
```
plot_train_val_acc(save_name='q2e', ret_dict=q2e_rd_128, orig_hp=ORIG_HPARAMS,
                   new_hp=q2e_hp_128, save='y', gru=False)
```



[68]:
```
q2e_rd_188, q2e_hp_188 = train_lab2e(embedding_dimension=188)
```

```
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 10,781,254 trainable parameters
Saving …
epoch: 1
train_loss: 0.502, train_acc: 0.752
valid_loss: 0.350, valid_acc: 0.859
Saving …
epoch: 2
train_loss: 0.229, train_acc: 0.912
valid_loss: 0.296, valid_acc: 0.882
Saving …
epoch: 3
train_loss: 0.110, train_acc: 0.962
valid_loss: 0.393, valid_acc: 0.881
Saving …
epoch: 4
train_loss: 0.047, train_acc: 0.984
valid_loss: 0.453, valid_acc: 0.876
Saving …
epoch: 5
train_loss: 0.023, train_acc: 0.992
valid_loss: 0.596, valid_acc: 0.874
test_loss: 0.604, test_acc: 0.874
```

[69]:
```
plot_train_val_acc(save_name='q2e', ret_dict=q2e_rd_188, orig_hp=ORIG_HPARAMS,
                   new_hp=q2e_hp_188, save='y', gru=False)
```

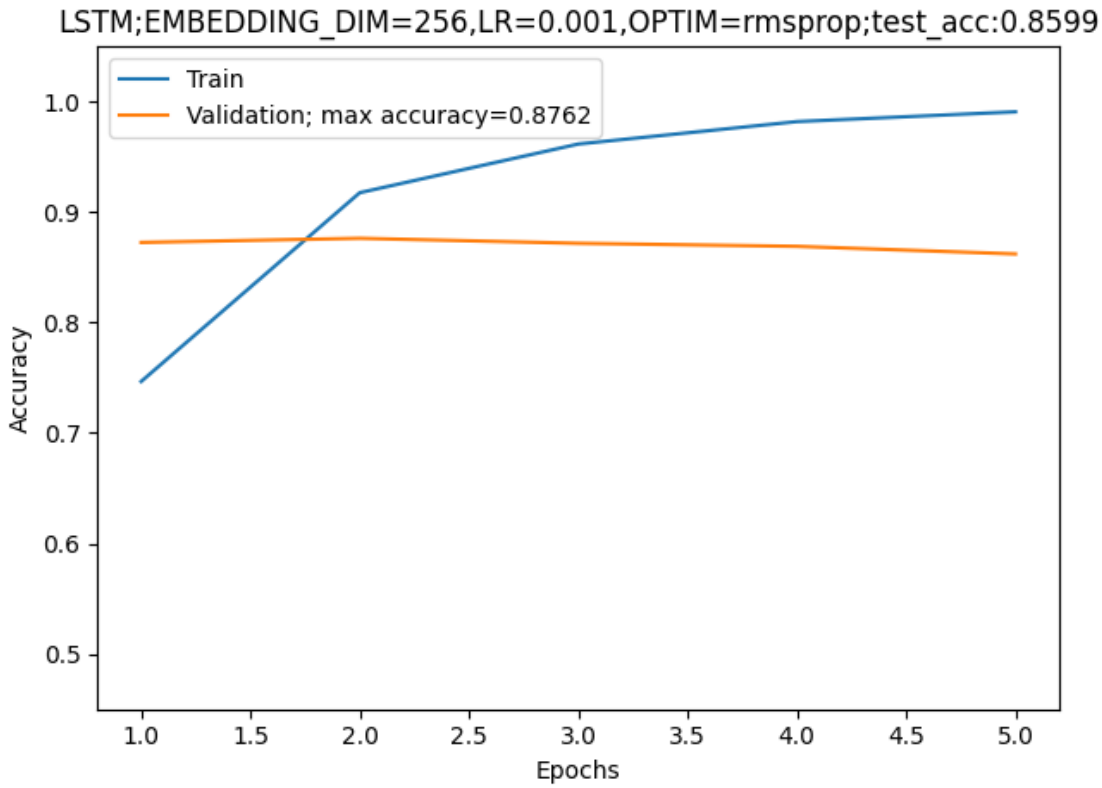


[70]:
```
q2e_rd_256, q2e_hp_256 = train_lab2e(embedding_dimension=256)
```

```
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 14,666,026 trainable parameters
Saving …
epoch: 1
train_loss: 0.511, train_acc: 0.747
valid_loss: 0.312, valid_acc: 0.872
Saving …
epoch: 2
train_loss: 0.215, train_acc: 0.917
valid_loss: 0.316, valid_acc: 0.876
Saving …
epoch: 3
train_loss: 0.107, train_acc: 0.962
valid_loss: 0.375, valid_acc: 0.872
Saving …
epoch: 4
train_loss: 0.052, train_acc: 0.982
```

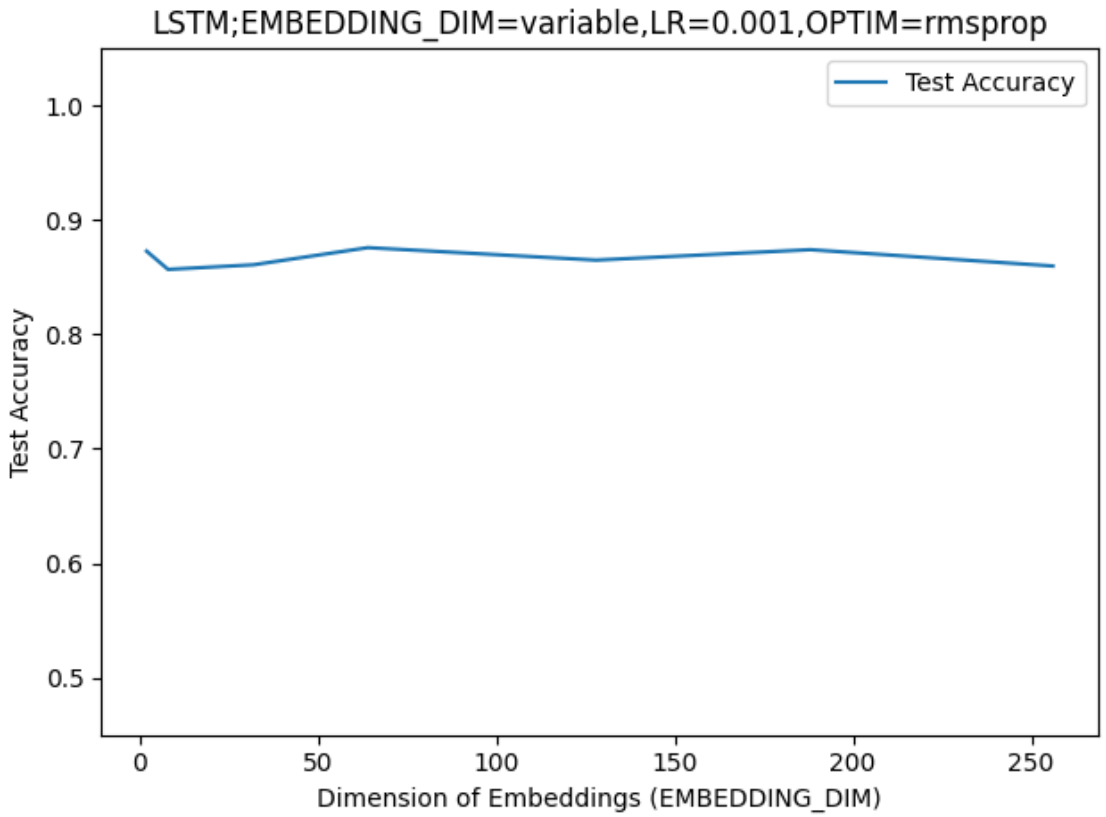
```
valid_loss: 0.480, valid_acc: 0.869
Saving …
epoch: 5
train_loss: 0.028, train_acc: 0.991
valid_loss: 0.618, valid_acc: 0.862
test_loss: 0.626, test_acc: 0.860
```

[71]:
```
plot_train_val_acc(save_name='q2e', ret_dict=q2e_rd_256, orig_hp=ORIG_HPARAMS,
                   new_hp=q2e_hp_256, save='y', gru=False)
```



[81]:
```
test_losses_q2e = [q2e_rd_2['test_acc'], q2e_rd_8['test_acc'],␣
 ↪q2e_rd_32['test_acc'],
                   q2e_rd_64['test_acc'], q2e_rd_128['test_acc'],␣
 ↪q2e_rd_188['test_acc'],
                   q2e_rd_256['test_acc']]
q2e_embedding_dim = [2, 8, 32, 64, 128, 188, 256]

fig, ax = plt.subplots(1, 1)

ax.plot(q2e_embedding_dim, test_losses_q2e, label='Test Accuracy')

ax.set_ylim([0.45, 1.05])
```

```
# diff_param = diff_hparams(orig_hp=orig_hp, new_hp=new_hp)

title_ = "LSTM;EMBEDDING_DIM=variable,LR=0.001,OPTIM=rmsprop"

ax.set_xlabel('Dimension of Embeddings (EMBEDDING_DIM)')
ax.set_ylabel('Test Accuracy')
ax.legend()
ax.set_title(title_)
fig.tight_layout()
plt.savefig('q2e_overall.pdf', dpi=500, bbox_inches='tight')
```



1.6.11 Lab 2(f) Compound scaling of embedding_dim, hidden_dim, layers

[83]:
```
def assert_all(hp_):
    assert_embedding_dim(hp_)
    assert_h_dim(hp_)
    assert hp_.N_LAYERS >= 1 and hp_.N_LAYERS <= 4
    return
```

```
def gen_compound_hparams():
    """
    returns list of hyperparam objects
    """
    # changes embedding_dim, hidden_dim, n_layers, and dropout
    hp_list = []
    num_epochs = 5
    learning_rate = 0.001
    which_optim = "rmsprop"


    lstm_nlayers = [1, 2, 3, 4]  # part c
    hidden_dim = [20, 280, 290, 305];  # part d
    embedding_table = [32, 188, 256]


    for idx_out, nlayer in enumerate(lstm_nlayers):
        for idx_mid, hid_dim in enumerate(hidden_dim):
            for idx_in, emb_dim in enumerate(embedding_table):
                temp_hp = HyperParams()
                temp_hp.N_EPOCHS = num_epochs
                temp_hp.LR = learning_rate
                temp_hp.OPTIM = which_optim

                temp_hp.N_LAYERS = nlayer
                temp_hp.HIDDEN_DIM = hid_dim
                temp_hp.EMBEDDING_DIM = emb_dim
                if nlayer >= 3 or hid_dim >= 250 or emb_dim >= 200:
                    temp_hp.DROPOUT_RATE = 0.2
                assert_all(hp_=temp_hp)
                diff_hparams(orig_hp=ORIG_HPARAMS, new_hp=temp_hp)
                hp_list.append(temp_hp)
    return hp_list


def train_compound_hparams(hp_compound_list):
    max_test_acc = -100
    hp_idx_for_max_test_acc = -100
    test_acc_list = []
    for idx_, hp_compound in enumerate(hp_compound_list):
        print('hparam', idx_)
        print(diff_hparams(orig_hp=ORIG_HPARAMS, new_hp=hp_compound))
        rd_compound = train_and_test_model_with_hparams(hparams=hp_compound,␣
↪model_type="lstm")
        test_acc_list.append(rd_compound["test_acc"])

        if rd_compound["test_acc"] > max_test_acc:
```

```
            max_test_acc = rd_compound["test_acc"]
            hp_idx_for_max_test_acc = idx_
        print("\n")
    print('max_test_acc', max_test_acc, '; which_hp', hp_idx_for_max_test_acc)
    print('diff_hp for hp that reached max_test_acc',
          diff_hparams(orig_hp=ORIG_HPARAMS,␣
 ↪new_hp=hp_compound_list[hp_idx_for_max_test_acc]))
    return test_acc_list


hp_list_out = gen_compound_hparams()
test_acc_list_compound = train_compound_hparams(hp_compound_list=hp_list_out)
```

```
hparam 0
EMBEDDING_DIM=32,HIDDEN_DIM=20,LR=0.001,OPTIM=rmsprop
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 1,819,690 trainable parameters
Saving …
epoch: 1
train_loss: 0.549, train_acc: 0.726
valid_loss: 0.335, valid_acc: 0.858
Saving …
epoch: 2
train_loss: 0.216, train_acc: 0.916
valid_loss: 0.278, valid_acc: 0.890
Saving …
epoch: 3
train_loss: 0.071, train_acc: 0.977
valid_loss: 0.372, valid_acc: 0.875
Saving …
epoch: 4
train_loss: 0.022, train_acc: 0.994
valid_loss: 0.482, valid_acc: 0.880
Saving …
epoch: 5
train_loss: 0.008, train_acc: 0.998
valid_loss: 0.559, valid_acc: 0.874
test_loss: 0.579, test_acc: 0.872


hparam 1
EMBEDDING_DIM=188,HIDDEN_DIM=20,LR=0.001,OPTIM=rmsprop
shape of train data is 35000
shape of test data is 10000
```

```
shape of valid data is 5000
Length of vocabulary is 56729
The model has 10,681,894 trainable parameters
Saving …
epoch: 1
train_loss: 0.549, train_acc: 0.724
valid_loss: 0.383, valid_acc: 0.839
Saving …
epoch: 2
train_loss: 0.236, train_acc: 0.910
valid_loss: 0.329, valid_acc: 0.874
Saving …
epoch: 3
train_loss: 0.088, train_acc: 0.969
valid_loss: 0.356, valid_acc: 0.874
Saving …
epoch: 4
train_loss: 0.033, train_acc: 0.989
valid_loss: 0.517, valid_acc: 0.866
Saving …
epoch: 5
train_loss: 0.016, train_acc: 0.995
valid_loss: 0.589, valid_acc: 0.873
test_loss: 0.632, test_acc: 0.863


hparam 2
DROPOUT_RATE=0.2,EMBEDDING_DIM=256,HIDDEN_DIM=20,LR=0.001,OPTIM=rmsprop
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 14,544,906 trainable parameters
Saving …
epoch: 1
train_loss: 0.552, train_acc: 0.726
valid_loss: 0.363, valid_acc: 0.858
Saving …
epoch: 2
train_loss: 0.270, train_acc: 0.901
valid_loss: 0.316, valid_acc: 0.869
Saving …
epoch: 3
train_loss: 0.142, train_acc: 0.953
valid_loss: 0.361, valid_acc: 0.868
Saving …
epoch: 4
train_loss: 0.076, train_acc: 0.975
```

```
valid_loss: 0.446, valid_acc: 0.863
Saving …
epoch: 5
train_loss: 0.042, train_acc: 0.987
valid_loss: 0.546, valid_acc: 0.864
test_loss: 0.570, test_acc: 0.858


hparam 3
DROPOUT_RATE=0.2,EMBEDDING_DIM=32,HIDDEN_DIM=280,LR=0.001,OPTIM=rmsprop
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 2,167,570 trainable parameters
Saving …
epoch: 1
train_loss: 0.721, train_acc: 0.542
valid_loss: 0.677, valid_acc: 0.568
Saving …
epoch: 2
train_loss: 0.643, train_acc: 0.629
valid_loss: 0.677, valid_acc: 0.594
Saving …
epoch: 3
train_loss: 0.582, train_acc: 0.688
valid_loss: 0.696, valid_acc: 0.604
Saving …
epoch: 4
train_loss: 0.541, train_acc: 0.719
valid_loss: 0.676, valid_acc: 0.619
Saving …
epoch: 5
train_loss: 0.490, train_acc: 0.756
valid_loss: 0.707, valid_acc: 0.638
test_loss: 0.692, test_acc: 0.636


hparam 4
DROPOUT_RATE=0.2,EMBEDDING_DIM=188,HIDDEN_DIM=280,LR=0.001,OPTIM=rmsprop
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 11,192,014 trainable parameters
Saving …
epoch: 1
train_loss: 0.488, train_acc: 0.761
```

```
valid_loss: 0.308, valid_acc: 0.870
Saving …
epoch: 2
train_loss: 0.232, train_acc: 0.909
valid_loss: 0.294, valid_acc: 0.885
Saving …
epoch: 3
train_loss: 0.108, train_acc: 0.964
valid_loss: 0.337, valid_acc: 0.881
Saving …
epoch: 4
train_loss: 0.055, train_acc: 0.982
valid_loss: 0.481, valid_acc: 0.876
Saving …
epoch: 5
train_loss: 0.032, train_acc: 0.989
valid_loss: 0.603, valid_acc: 0.866
test_loss: 0.652, test_acc: 0.858


hparam 5
DROPOUT_RATE=0.2,EMBEDDING_DIM=256,HIDDEN_DIM=280,LR=0.001,OPTIM=rmsprop
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 15,125,746 trainable parameters
Saving …
epoch: 1
train_loss: 0.510, train_acc: 0.752
valid_loss: 0.312, valid_acc: 0.876
Saving …
epoch: 2
train_loss: 0.210, train_acc: 0.921
valid_loss: 0.295, valid_acc: 0.882
Saving …
epoch: 3
train_loss: 0.110, train_acc: 0.961
valid_loss: 0.425, valid_acc: 0.873
Saving …
epoch: 4
train_loss: 0.067, train_acc: 0.976
valid_loss: 0.412, valid_acc: 0.867
Saving …
epoch: 5
train_loss: 0.037, train_acc: 0.988
valid_loss: 0.600, valid_acc: 0.868
test_loss: 0.617, test_acc: 0.863
```

```
hparam 6
DROPOUT_RATE=0.2,EMBEDDING_DIM=32,HIDDEN_DIM=290,LR=0.001,OPTIM=rmsprop
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 2,191,750 trainable parameters
Saving …
epoch: 1
train_loss: 0.709, train_acc: 0.579
valid_loss: 0.694, valid_acc: 0.574
Saving …
epoch: 2
train_loss: 0.468, train_acc: 0.782
valid_loss: 0.411, valid_acc: 0.816
Saving …
epoch: 3
train_loss: 0.310, train_acc: 0.873
valid_loss: 0.353, valid_acc: 0.851
Saving …
epoch: 4
train_loss: 0.198, train_acc: 0.926
valid_loss: 0.343, valid_acc: 0.870
Saving …
epoch: 5
train_loss: 0.139, train_acc: 0.951
valid_loss: 0.355, valid_acc: 0.864
test_loss: 0.364, test_acc: 0.859


hparam 7
DROPOUT_RATE=0.2,EMBEDDING_DIM=188,HIDDEN_DIM=290,LR=0.001,OPTIM=rmsprop
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 11,222,434 trainable parameters
Saving …
epoch: 1
train_loss: 0.507, train_acc: 0.755
valid_loss: 0.378, valid_acc: 0.844
Saving …
epoch: 2
train_loss: 0.233, train_acc: 0.909
valid_loss: 0.312, valid_acc: 0.868
Saving …
```

```
epoch: 3
train_loss: 0.111, train_acc: 0.962
valid_loss: 0.351, valid_acc: 0.873
Saving …
epoch: 4
train_loss: 0.057, train_acc: 0.981
valid_loss: 0.481, valid_acc: 0.873
Saving …
epoch: 5
train_loss: 0.034, train_acc: 0.988
valid_loss: 0.639, valid_acc: 0.875
test_loss: 0.666, test_acc: 0.869


hparam 8
DROPOUT_RATE=0.2,EMBEDDING_DIM=256,HIDDEN_DIM=290,LR=0.001,OPTIM=rmsprop
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 15,158,886 trainable parameters
Saving …
epoch: 1
train_loss: 0.504, train_acc: 0.752
valid_loss: 0.313, valid_acc: 0.869
Saving …
epoch: 2
train_loss: 0.217, train_acc: 0.917
valid_loss: 0.318, valid_acc: 0.884
Saving …
epoch: 3
train_loss: 0.115, train_acc: 0.959
valid_loss: 0.361, valid_acc: 0.869
Saving …
epoch: 4
train_loss: 0.065, train_acc: 0.978
valid_loss: 0.410, valid_acc: 0.862
Saving …
epoch: 5
train_loss: 0.043, train_acc: 0.985
valid_loss: 0.574, valid_acc: 0.863
test_loss: 0.600, test_acc: 0.865


hparam 9
DROPOUT_RATE=0.2,EMBEDDING_DIM=32,HIDDEN_DIM=305,LR=0.001,OPTIM=rmsprop
shape of train data is 35000
shape of test data is 10000
```

```
shape of valid data is 5000
Length of vocabulary is 56729
The model has 2,229,520 trainable parameters
Saving …
epoch: 1
train_loss: 0.533, train_acc: 0.736
valid_loss: 0.575, valid_acc: 0.677
Saving …
epoch: 2
train_loss: 0.251, train_acc: 0.900
valid_loss: 0.332, valid_acc: 0.873
Saving …
epoch: 3
train_loss: 0.128, train_acc: 0.954
valid_loss: 0.333, valid_acc: 0.881
Saving …
epoch: 4
train_loss: 0.057, train_acc: 0.982
valid_loss: 0.434, valid_acc: 0.875
Saving …
epoch: 5
train_loss: 0.027, train_acc: 0.992
valid_loss: 0.678, valid_acc: 0.866
test_loss: 0.701, test_acc: 0.861


hparam 10
DROPOUT_RATE=0.2,EMBEDDING_DIM=188,HIDDEN_DIM=305,LR=0.001,OPTIM=rmsprop
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 11,269,564 trainable parameters
Saving …
epoch: 1
train_loss: 0.542, train_acc: 0.758
valid_loss: 0.323, valid_acc: 0.863
Saving …
epoch: 2
train_loss: 0.217, train_acc: 0.917
valid_loss: 0.300, valid_acc: 0.878
Saving …
epoch: 3
train_loss: 0.101, train_acc: 0.965
valid_loss: 0.374, valid_acc: 0.877
Saving …
epoch: 4
train_loss: 0.049, train_acc: 0.984
```

```
valid_loss: 0.444, valid_acc: 0.874
Saving …
epoch: 5
train_loss: 0.029, train_acc: 0.991
valid_loss: 0.576, valid_acc: 0.876
test_loss: 0.610, test_acc: 0.875


hparam 11
DROPOUT_RATE=0.2,EMBEDDING_DIM=256,HIDDEN_DIM=305,LR=0.001,OPTIM=rmsprop
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 15,210,096 trainable parameters
Saving …
epoch: 1
train_loss: 0.509, train_acc: 0.757
valid_loss: 0.323, valid_acc: 0.867
Saving …
epoch: 2
train_loss: 0.211, train_acc: 0.919
valid_loss: 0.328, valid_acc: 0.884
Saving …
epoch: 3
train_loss: 0.103, train_acc: 0.964
valid_loss: 0.393, valid_acc: 0.877
Saving …
epoch: 4
train_loss: 0.057, train_acc: 0.981
valid_loss: 0.464, valid_acc: 0.870
Saving …
epoch: 5
train_loss: 0.037, train_acc: 0.988
valid_loss: 0.581, valid_acc: 0.870
test_loss: 0.596, test_acc: 0.861


hparam 12
EMBEDDING_DIM=32,HIDDEN_DIM=20,LR=0.001,N_LAYERS=2,OPTIM=rmsprop
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 1,823,050 trainable parameters
Saving …
epoch: 1
train_loss: 0.499, train_acc: 0.758
```

```
valid_loss: 0.406, valid_acc: 0.827
Saving …
epoch: 2
train_loss: 0.310, train_acc: 0.873
valid_loss: 0.340, valid_acc: 0.856
Saving …
epoch: 3
train_loss: 0.179, train_acc: 0.934
valid_loss: 0.331, valid_acc: 0.869
Saving …
epoch: 4
train_loss: 0.090, train_acc: 0.968
valid_loss: 0.332, valid_acc: 0.878
Saving …
epoch: 5
train_loss: 0.039, train_acc: 0.988
valid_loss: 0.463, valid_acc: 0.873
test_loss: 0.492, test_acc: 0.869


hparam 13
EMBEDDING_DIM=188,HIDDEN_DIM=20,LR=0.001,N_LAYERS=2,OPTIM=rmsprop
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 10,685,254 trainable parameters
Saving …
epoch: 1
train_loss: 0.475, train_acc: 0.778
valid_loss: 0.341, valid_acc: 0.863
Saving …
epoch: 2
train_loss: 0.241, train_acc: 0.908
valid_loss: 0.332, valid_acc: 0.870
Saving …
epoch: 3
train_loss: 0.138, train_acc: 0.952
valid_loss: 0.385, valid_acc: 0.861
Saving …
epoch: 4
train_loss: 0.074, train_acc: 0.975
valid_loss: 0.398, valid_acc: 0.869
Saving …
epoch: 5
train_loss: 0.039, train_acc: 0.986
valid_loss: 0.507, valid_acc: 0.858
test_loss: 0.558, test_acc: 0.853
```

```
hparam 14
DROPOUT_RATE=0.2,EMBEDDING_DIM=256,HIDDEN_DIM=20,LR=0.001,N_LAYERS=2,OPTIM=rmspr
op
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 14,548,266 trainable parameters
Saving …
epoch: 1
train_loss: 0.547, train_acc: 0.719
valid_loss: 0.329, valid_acc: 0.866
Saving …
epoch: 2
train_loss: 0.229, train_acc: 0.913
valid_loss: 0.270, valid_acc: 0.889
Saving …
epoch: 3
train_loss: 0.086, train_acc: 0.972
valid_loss: 0.391, valid_acc: 0.872
Saving …
epoch: 4
train_loss: 0.036, train_acc: 0.989
valid_loss: 0.409, valid_acc: 0.872
Saving …
epoch: 5
train_loss: 0.020, train_acc: 0.994
valid_loss: 0.589, valid_acc: 0.864
test_loss: 0.586, test_acc: 0.867


hparam 15
DROPOUT_RATE=0.2,EMBEDDING_DIM=32,HIDDEN_DIM=280,LR=0.001,N_LAYERS=2,OPTIM=rmspr
op
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 2,797,010 trainable parameters
Saving …
epoch: 1
train_loss: 0.523, train_acc: 0.733
valid_loss: 0.490, valid_acc: 0.815
Saving …
epoch: 2
train_loss: 0.270, train_acc: 0.895
```

```
valid_loss: 0.334, valid_acc: 0.864
Saving …
epoch: 3
train_loss: 0.179, train_acc: 0.933
valid_loss: 0.322, valid_acc: 0.871
Saving …
epoch: 4
train_loss: 0.125, train_acc: 0.957
valid_loss: 0.348, valid_acc: 0.871
Saving …
epoch: 5
train_loss: 0.083, train_acc: 0.973
valid_loss: 0.382, valid_acc: 0.876
test_loss: 0.390, test_acc: 0.872


hparam 16
DROPOUT_RATE=0.2,EMBEDDING_DIM=188,HIDDEN_DIM=280,LR=0.001,N_LAYERS=2,OPTIM=rmsp
rop
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 11,821,454 trainable parameters
Saving …
epoch: 1
train_loss: 0.494, train_acc: 0.758
valid_loss: 0.308, valid_acc: 0.876
Saving …
epoch: 2
train_loss: 0.231, train_acc: 0.912
valid_loss: 0.303, valid_acc: 0.872
Saving …
epoch: 3
train_loss: 0.131, train_acc: 0.955
valid_loss: 0.358, valid_acc: 0.881
Saving …
epoch: 4
train_loss: 0.074, train_acc: 0.977
valid_loss: 0.417, valid_acc: 0.879
Saving …
epoch: 5
train_loss: 0.044, train_acc: 0.985
valid_loss: 0.570, valid_acc: 0.873
test_loss: 0.621, test_acc: 0.870


hparam 17
```

```
DROPOUT_RATE=0.2,EMBEDDING_DIM=256,HIDDEN_DIM=280,LR=0.001,N_LAYERS=2,OPTIM=rmsp
rop
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 15,755,186 trainable parameters
Saving …
epoch: 1
train_loss: 0.491, train_acc: 0.756
valid_loss: 0.309, valid_acc: 0.866
Saving …
epoch: 2
train_loss: 0.239, train_acc: 0.908
valid_loss: 0.309, valid_acc: 0.881
Saving …
epoch: 3
train_loss: 0.155, train_acc: 0.944
valid_loss: 0.306, valid_acc: 0.874
Saving …
epoch: 4
train_loss: 0.100, train_acc: 0.967
valid_loss: 0.365, valid_acc: 0.886
Saving …
epoch: 5
train_loss: 0.070, train_acc: 0.978
valid_loss: 0.441, valid_acc: 0.878
test_loss: 0.449, test_acc: 0.873


hparam 18
DROPOUT_RATE=0.2,EMBEDDING_DIM=32,HIDDEN_DIM=290,LR=0.001,N_LAYERS=2,OPTIM=rmspr
op
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 2,866,870 trainable parameters
Saving …
epoch: 1
train_loss: 0.667, train_acc: 0.577
valid_loss: 0.495, valid_acc: 0.761
Saving …
epoch: 2
train_loss: 0.347, train_acc: 0.855
valid_loss: 0.298, valid_acc: 0.876
Saving …
epoch: 3
```

```
train_loss: 0.187, train_acc: 0.932
valid_loss: 0.357, valid_acc: 0.880
Saving …
epoch: 4
train_loss: 0.111, train_acc: 0.963
valid_loss: 0.317, valid_acc: 0.887
Saving …
epoch: 5
train_loss: 0.066, train_acc: 0.978
valid_loss: 0.445, valid_acc: 0.874
test_loss: 0.442, test_acc: 0.871


hparam 19
DROPOUT_RATE=0.2,EMBEDDING_DIM=188,HIDDEN_DIM=290,LR=0.001,N_LAYERS=2,OPTIM=rmsp
rop
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 11,897,554 trainable parameters
Saving …
epoch: 1
train_loss: 0.605, train_acc: 0.677
valid_loss: 0.399, valid_acc: 0.826
Saving …
epoch: 2
train_loss: 0.319, train_acc: 0.868
valid_loss: 0.326, valid_acc: 0.861
Saving …
epoch: 3
train_loss: 0.217, train_acc: 0.918
valid_loss: 0.319, valid_acc: 0.860
Saving …
epoch: 4
train_loss: 0.146, train_acc: 0.949
valid_loss: 0.399, valid_acc: 0.855
Saving …
epoch: 5
train_loss: 0.102, train_acc: 0.965
valid_loss: 0.415, valid_acc: 0.859
test_loss: 0.421, test_acc: 0.860


hparam 20
DROPOUT_RATE=0.2,EMBEDDING_DIM=256,HIDDEN_DIM=290,LR=0.001,N_LAYERS=2,OPTIM=rmsp
rop
shape of train data is 35000
```

```
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 15,834,006 trainable parameters
Saving …
epoch: 1
train_loss: 0.526, train_acc: 0.739
valid_loss: 0.385, valid_acc: 0.830
Saving …
epoch: 2
train_loss: 0.256, train_acc: 0.899
valid_loss: 0.306, valid_acc: 0.874
Saving …
epoch: 3
train_loss: 0.156, train_acc: 0.944
valid_loss: 0.342, valid_acc: 0.880
Saving …
epoch: 4
train_loss: 0.102, train_acc: 0.966
valid_loss: 0.348, valid_acc: 0.878
Saving …
epoch: 5
train_loss: 0.061, train_acc: 0.981
valid_loss: 0.430, valid_acc: 0.876
test_loss: 0.455, test_acc: 0.872


hparam 21
DROPOUT_RATE=0.2,EMBEDDING_DIM=32,HIDDEN_DIM=305,LR=0.001,N_LAYERS=2,OPTIM=rmspr
op
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 2,976,160 trainable parameters
Saving …
epoch: 1
train_loss: 0.639, train_acc: 0.646
valid_loss: 0.598, valid_acc: 0.682
Saving …
epoch: 2
train_loss: 0.344, train_acc: 0.854
valid_loss: 0.306, valid_acc: 0.870
Saving …
epoch: 3
train_loss: 0.194, train_acc: 0.929
valid_loss: 0.368, valid_acc: 0.856
Saving …
```

```
epoch: 4
train_loss: 0.117, train_acc: 0.960
valid_loss: 0.320, valid_acc: 0.875
Saving …
epoch: 5
train_loss: 0.055, train_acc: 0.983
valid_loss: 0.419, valid_acc: 0.882
test_loss: 0.441, test_acc: 0.880


hparam 22
DROPOUT_RATE=0.2,EMBEDDING_DIM=188,HIDDEN_DIM=305,LR=0.001,N_LAYERS=2,OPTIM=rmsp
rop
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 12,016,204 trainable parameters
Saving …
epoch: 1
train_loss: 0.530, train_acc: 0.728
valid_loss: 0.308, valid_acc: 0.870
Saving …
epoch: 2
train_loss: 0.239, train_acc: 0.908
valid_loss: 0.301, valid_acc: 0.882
Saving …
epoch: 3
train_loss: 0.132, train_acc: 0.952
valid_loss: 0.330, valid_acc: 0.878
Saving …
epoch: 4
train_loss: 0.073, train_acc: 0.977
valid_loss: 0.437, valid_acc: 0.878
Saving …
epoch: 5
train_loss: 0.048, train_acc: 0.985
valid_loss: 0.548, valid_acc: 0.857
test_loss: 0.573, test_acc: 0.849


hparam 23
DROPOUT_RATE=0.2,EMBEDDING_DIM=256,HIDDEN_DIM=305,LR=0.001,N_LAYERS=2,OPTIM=rmsp
rop
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
```

```
The model has 15,956,736 trainable parameters
Saving …
epoch: 1
train_loss: 0.721, train_acc: 0.514
valid_loss: 0.673, valid_acc: 0.606
Saving …
epoch: 2
train_loss: 0.474, train_acc: 0.780
valid_loss: 0.387, valid_acc: 0.831
Saving …
epoch: 3
train_loss: 0.278, train_acc: 0.890
valid_loss: 0.357, valid_acc: 0.858
Saving …
epoch: 4
train_loss: 0.201, train_acc: 0.927
valid_loss: 0.355, valid_acc: 0.862
Saving …
epoch: 5
train_loss: 0.144, train_acc: 0.949
valid_loss: 0.358, valid_acc: 0.877
test_loss: 0.389, test_acc: 0.865


hparam 24
DROPOUT_RATE=0.2,EMBEDDING_DIM=32,HIDDEN_DIM=20,LR=0.001,N_LAYERS=3,OPTIM=rmspro
p
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 1,826,410 trainable parameters
Saving …
epoch: 1
train_loss: 0.658, train_acc: 0.599
valid_loss: 0.623, valid_acc: 0.669
Saving …
epoch: 2
train_loss: 0.402, train_acc: 0.824
valid_loss: 0.301, valid_acc: 0.875
Saving …
epoch: 3
train_loss: 0.198, train_acc: 0.928
valid_loss: 0.285, valid_acc: 0.886
Saving …
epoch: 4
train_loss: 0.105, train_acc: 0.965
valid_loss: 0.298, valid_acc: 0.888
```

```
Saving …
epoch: 5
train_loss: 0.054, train_acc: 0.983
valid_loss: 0.424, valid_acc: 0.874
test_loss: 0.448, test_acc: 0.869


hparam 25
DROPOUT_RATE=0.2,EMBEDDING_DIM=188,HIDDEN_DIM=20,LR=0.001,N_LAYERS=3,OPTIM=rmspr
op
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 10,688,614 trainable parameters
Saving …
epoch: 1
train_loss: 0.581, train_acc: 0.698
valid_loss: 0.428, valid_acc: 0.810
Saving …
epoch: 2
train_loss: 0.313, train_acc: 0.873
valid_loss: 0.322, valid_acc: 0.872
Saving …
epoch: 3
train_loss: 0.178, train_acc: 0.935
valid_loss: 0.310, valid_acc: 0.879
Saving …
epoch: 4
train_loss: 0.106, train_acc: 0.963
valid_loss: 0.371, valid_acc: 0.871
Saving …
epoch: 5
train_loss: 0.060, train_acc: 0.981
valid_loss: 0.416, valid_acc: 0.873
test_loss: 0.447, test_acc: 0.862


hparam 26
DROPOUT_RATE=0.2,EMBEDDING_DIM=256,HIDDEN_DIM=20,LR=0.001,N_LAYERS=3,OPTIM=rmspr
op
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 14,551,626 trainable parameters
Saving …
epoch: 1
```

```
train_loss: 0.543, train_acc: 0.724
valid_loss: 0.365, valid_acc: 0.841
Saving …
epoch: 2
train_loss: 0.265, train_acc: 0.899
valid_loss: 0.302, valid_acc: 0.881
Saving …
epoch: 3
train_loss: 0.130, train_acc: 0.955
valid_loss: 0.327, valid_acc: 0.874
Saving …
epoch: 4
train_loss: 0.068, train_acc: 0.978
valid_loss: 0.423, valid_acc: 0.880
Saving …
epoch: 5
train_loss: 0.032, train_acc: 0.990
valid_loss: 0.434, valid_acc: 0.871
test_loss: 0.437, test_acc: 0.866


hparam 27
DROPOUT_RATE=0.2,EMBEDDING_DIM=32,HIDDEN_DIM=280,LR=0.001,N_LAYERS=3,OPTIM=rmspr
op
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 3,426,450 trainable parameters
Saving …
epoch: 1
train_loss: 0.699, train_acc: 0.494
valid_loss: 0.693, valid_acc: 0.501
Saving …
epoch: 2
train_loss: 0.694, train_acc: 0.497
valid_loss: 0.693, valid_acc: 0.501
Saving …
epoch: 3
train_loss: 0.693, train_acc: 0.498
valid_loss: 0.693, valid_acc: 0.499
Saving …
epoch: 4
train_loss: 0.694, train_acc: 0.501
valid_loss: 0.693, valid_acc: 0.501
Saving …
epoch: 5
train_loss: 0.699, train_acc: 0.507
```

```
valid_loss: 0.691, valid_acc: 0.499
test_loss: 0.690, test_acc: 0.500


hparam 28
DROPOUT_RATE=0.2,EMBEDDING_DIM=188,HIDDEN_DIM=280,LR=0.001,N_LAYERS=3,OPTIM=rmsp
rop
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 12,450,894 trainable parameters
Saving …
epoch: 1
train_loss: 0.626, train_acc: 0.622
valid_loss: 0.378, valid_acc: 0.844
Saving …
epoch: 2
train_loss: 0.325, train_acc: 0.868
valid_loss: 0.304, valid_acc: 0.878
Saving …
epoch: 3
train_loss: 0.231, train_acc: 0.917
valid_loss: 0.366, valid_acc: 0.857
Saving …
epoch: 4
train_loss: 0.152, train_acc: 0.949
valid_loss: 0.346, valid_acc: 0.867
Saving …
epoch: 5
train_loss: 0.109, train_acc: 0.966
valid_loss: 0.371, valid_acc: 0.867
test_loss: 0.399, test_acc: 0.864


hparam 29
DROPOUT_RATE=0.2,EMBEDDING_DIM=256,HIDDEN_DIM=280,LR=0.001,N_LAYERS=3,OPTIM=rmsp
rop
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 16,384,626 trainable parameters
Saving …
epoch: 1
train_loss: 0.681, train_acc: 0.562
valid_loss: 0.666, valid_acc: 0.566
Saving …
```

```
epoch: 2
train_loss: 0.403, train_acc: 0.824
valid_loss: 0.328, valid_acc: 0.867
Saving …
epoch: 3
train_loss: 0.249, train_acc: 0.909
valid_loss: 0.463, valid_acc: 0.842
Saving …
epoch: 4
train_loss: 0.184, train_acc: 0.937
valid_loss: 0.395, valid_acc: 0.865
Saving …
epoch: 5
train_loss: 0.133, train_acc: 0.956
valid_loss: 0.350, valid_acc: 0.880
test_loss: 0.374, test_acc: 0.871


hparam 30
DROPOUT_RATE=0.2,EMBEDDING_DIM=32,HIDDEN_DIM=290,LR=0.001,N_LAYERS=3,OPTIM=rmspr
op
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 3,541,990 trainable parameters
Saving …
epoch: 1
train_loss: 0.695, train_acc: 0.560
valid_loss: 0.684, valid_acc: 0.535
Saving …
epoch: 2
train_loss: 0.371, train_acc: 0.841
valid_loss: 0.311, valid_acc: 0.861
Saving …
epoch: 3
train_loss: 0.208, train_acc: 0.922
valid_loss: 0.298, valid_acc: 0.881
Saving …
epoch: 4
train_loss: 0.141, train_acc: 0.952
valid_loss: 0.333, valid_acc: 0.882
Saving …
epoch: 5
train_loss: 0.098, train_acc: 0.969
valid_loss: 0.341, valid_acc: 0.871
test_loss: 0.350, test_acc: 0.867
```

```
hparam 31
DROPOUT_RATE=0.2,EMBEDDING_DIM=188,HIDDEN_DIM=290,LR=0.001,N_LAYERS=3,OPTIM=rmsp
rop
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 12,572,674 trainable parameters
Saving …
epoch: 1
train_loss: 0.680, train_acc: 0.548
valid_loss: 0.693, valid_acc: 0.501
Saving …
epoch: 2
train_loss: 0.575, train_acc: 0.658
valid_loss: 0.417, valid_acc: 0.799
Saving …
epoch: 3
train_loss: 0.299, train_acc: 0.883
valid_loss: 0.323, valid_acc: 0.870
Saving …
epoch: 4
train_loss: 0.205, train_acc: 0.925
valid_loss: 0.330, valid_acc: 0.872
Saving …
epoch: 5
train_loss: 0.153, train_acc: 0.947
valid_loss: 0.349, valid_acc: 0.876
test_loss: 0.362, test_acc: 0.872


hparam 32
DROPOUT_RATE=0.2,EMBEDDING_DIM=256,HIDDEN_DIM=290,LR=0.001,N_LAYERS=3,OPTIM=rmsp
rop
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 16,509,126 trainable parameters
Saving …
epoch: 1
train_loss: 0.504, train_acc: 0.753
valid_loss: 0.369, valid_acc: 0.842
Saving …
epoch: 2
train_loss: 0.266, train_acc: 0.898
valid_loss: 0.296, valid_acc: 0.883
```

```
Saving …
epoch: 3
train_loss: 0.186, train_acc: 0.933
valid_loss: 0.365, valid_acc: 0.874
Saving …
epoch: 4
train_loss: 0.133, train_acc: 0.954
valid_loss: 0.359, valid_acc: 0.873
Saving …
epoch: 5
train_loss: 0.098, train_acc: 0.969
valid_loss: 0.401, valid_acc: 0.870
test_loss: 0.419, test_acc: 0.866


hparam 33
DROPOUT_RATE=0.2,EMBEDDING_DIM=32,HIDDEN_DIM=305,LR=0.001,N_LAYERS=3,OPTIM=rmspr
op
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 3,722,800 trainable parameters
Saving …
epoch: 1
train_loss: 0.686, train_acc: 0.567
valid_loss: 0.676, valid_acc: 0.590
Saving …
epoch: 2
train_loss: 0.682, train_acc: 0.573
valid_loss: 0.678, valid_acc: 0.594
Saving …
epoch: 3
train_loss: 0.579, train_acc: 0.684
valid_loss: 0.451, valid_acc: 0.788
Saving …
epoch: 4
train_loss: 0.362, train_acc: 0.852
valid_loss: 0.361, valid_acc: 0.859
Saving …
epoch: 5
train_loss: 0.244, train_acc: 0.909
valid_loss: 0.300, valid_acc: 0.874
test_loss: 0.317, test_acc: 0.869


hparam 34
DROPOUT_RATE=0.2,EMBEDDING_DIM=188,HIDDEN_DIM=305,LR=0.001,N_LAYERS=3,OPTIM=rmsp
```

```
rop
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 12,762,844 trainable parameters
Saving …
epoch: 1
train_loss: 0.575, train_acc: 0.679
valid_loss: 0.373, valid_acc: 0.845
Saving …
epoch: 2
train_loss: 0.292, train_acc: 0.885
valid_loss: 0.317, valid_acc: 0.860
Saving …
epoch: 3
train_loss: 0.182, train_acc: 0.934
valid_loss: 0.322, valid_acc: 0.871
Saving …
epoch: 4
train_loss: 0.120, train_acc: 0.959
valid_loss: 0.349, valid_acc: 0.875
Saving …
epoch: 5
train_loss: 0.084, train_acc: 0.974
valid_loss: 0.550, valid_acc: 0.877
test_loss: 0.590, test_acc: 0.869


hparam 35
DROPOUT_RATE=0.2,EMBEDDING_DIM=256,HIDDEN_DIM=305,LR=0.001,N_LAYERS=3,OPTIM=rmsp
rop
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 16,703,376 trainable parameters
Saving …
epoch: 1
train_loss: 0.523, train_acc: 0.739
valid_loss: 0.378, valid_acc: 0.840
Saving …
epoch: 2
train_loss: 0.280, train_acc: 0.890
valid_loss: 0.343, valid_acc: 0.859
Saving …
epoch: 3
train_loss: 0.173, train_acc: 0.936
```

```
valid_loss: 0.356, valid_acc: 0.865
Saving …
epoch: 4
train_loss: 0.122, train_acc: 0.959
valid_loss: 0.357, valid_acc: 0.858
Saving …
epoch: 5
train_loss: 0.091, train_acc: 0.971
valid_loss: 0.439, valid_acc: 0.863
test_loss: 0.449, test_acc: 0.865


hparam 36
DROPOUT_RATE=0.2,EMBEDDING_DIM=32,HIDDEN_DIM=20,LR=0.001,N_LAYERS=4,OPTIM=rmspro
p
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 1,829,770 trainable parameters
Saving …
epoch: 1
train_loss: 0.602, train_acc: 0.662
valid_loss: 0.431, valid_acc: 0.797
Saving …
epoch: 2
train_loss: 0.323, train_acc: 0.869
valid_loss: 0.354, valid_acc: 0.855
Saving …
epoch: 3
train_loss: 0.173, train_acc: 0.938
valid_loss: 0.322, valid_acc: 0.881
Saving …
epoch: 4
train_loss: 0.091, train_acc: 0.971
valid_loss: 0.375, valid_acc: 0.882
Saving …
epoch: 5
train_loss: 0.050, train_acc: 0.985
valid_loss: 0.520, valid_acc: 0.879
test_loss: 0.546, test_acc: 0.872


hparam 37
DROPOUT_RATE=0.2,EMBEDDING_DIM=188,HIDDEN_DIM=20,LR=0.001,N_LAYERS=4,OPTIM=rmspr
op
shape of train data is 35000
shape of test data is 10000
```

```
shape of valid data is 5000
Length of vocabulary is 56729
The model has 10,691,974 trainable parameters
Saving …
epoch: 1
train_loss: 0.671, train_acc: 0.596
valid_loss: 0.639, valid_acc: 0.630
Saving …
epoch: 2
train_loss: 0.477, train_acc: 0.777
valid_loss: 0.336, valid_acc: 0.863
Saving …
epoch: 3
train_loss: 0.250, train_acc: 0.905
valid_loss: 0.305, valid_acc: 0.873
Saving …
epoch: 4
train_loss: 0.129, train_acc: 0.958
valid_loss: 0.343, valid_acc: 0.875
Saving …
epoch: 5
train_loss: 0.074, train_acc: 0.978
valid_loss: 0.394, valid_acc: 0.871
test_loss: 0.410, test_acc: 0.868


hparam 38
DROPOUT_RATE=0.2,EMBEDDING_DIM=256,HIDDEN_DIM=20,LR=0.001,N_LAYERS=4,OPTIM=rmspr
op
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 14,554,986 trainable parameters
Saving …
epoch: 1
train_loss: 0.639, train_acc: 0.605
valid_loss: 0.447, valid_acc: 0.809
Saving …
epoch: 2
train_loss: 0.331, train_acc: 0.866
valid_loss: 0.339, valid_acc: 0.860
Saving …
epoch: 3
train_loss: 0.180, train_acc: 0.936
valid_loss: 0.338, valid_acc: 0.873
Saving …
epoch: 4
```

```
train_loss: 0.098, train_acc: 0.968
valid_loss: 0.344, valid_acc: 0.882
Saving …
epoch: 5
train_loss: 0.050, train_acc: 0.985
valid_loss: 0.461, valid_acc: 0.876
test_loss: 0.507, test_acc: 0.870


hparam 39
DROPOUT_RATE=0.2,EMBEDDING_DIM=32,HIDDEN_DIM=280,LR=0.001,N_LAYERS=4,OPTIM=rmspr
op
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 4,055,890 trainable parameters
Saving …
epoch: 1
train_loss: 0.695, train_acc: 0.495
valid_loss: 0.693, valid_acc: 0.501
Saving …
epoch: 2
train_loss: 0.694, train_acc: 0.501
valid_loss: 0.693, valid_acc: 0.501
Saving …
epoch: 3
train_loss: 0.694, train_acc: 0.498
valid_loss: 0.693, valid_acc: 0.499
Saving …
epoch: 4
train_loss: 0.693, train_acc: 0.493
valid_loss: 0.693, valid_acc: 0.499
Saving …
epoch: 5
train_loss: 0.694, train_acc: 0.497
valid_loss: 0.694, valid_acc: 0.499
test_loss: 0.694, test_acc: 0.500


hparam 40
DROPOUT_RATE=0.2,EMBEDDING_DIM=188,HIDDEN_DIM=280,LR=0.001,N_LAYERS=4,OPTIM=rmsp
rop
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 13,080,334 trainable parameters
```

```
Saving …
epoch: 1
train_loss: 0.562, train_acc: 0.685
valid_loss: 0.354, valid_acc: 0.844
Saving …
epoch: 2
train_loss: 0.288, train_acc: 0.888
valid_loss: 0.299, valid_acc: 0.877
Saving …
epoch: 3
train_loss: 0.185, train_acc: 0.936
valid_loss: 0.295, valid_acc: 0.880
Saving …
epoch: 4
train_loss: 0.119, train_acc: 0.961
valid_loss: 0.358, valid_acc: 0.881
Saving …
epoch: 5
train_loss: 0.079, train_acc: 0.976
valid_loss: 0.428, valid_acc: 0.878
test_loss: 0.444, test_acc: 0.871


hparam 41
DROPOUT_RATE=0.2,EMBEDDING_DIM=256,HIDDEN_DIM=280,LR=0.001,N_LAYERS=4,OPTIM=rmsp
rop
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 17,014,066 trainable parameters
Saving …
epoch: 1
train_loss: 0.695, train_acc: 0.501
valid_loss: 0.693, valid_acc: 0.501
Saving …
epoch: 2
train_loss: 0.693, train_acc: 0.500
valid_loss: 0.693, valid_acc: 0.501
Saving …
epoch: 3
train_loss: 0.693, train_acc: 0.501
valid_loss: 0.694, valid_acc: 0.501
Saving …
epoch: 4
train_loss: 0.694, train_acc: 0.497
valid_loss: 0.693, valid_acc: 0.501
Saving …
```

```
epoch: 5
train_loss: 0.694, train_acc: 0.500
valid_loss: 0.693, valid_acc: 0.501
test_loss: 0.693, test_acc: 0.500


hparam 42
DROPOUT_RATE=0.2,EMBEDDING_DIM=32,HIDDEN_DIM=290,LR=0.001,N_LAYERS=4,OPTIM=rmspr
op
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 4,217,110 trainable parameters
Saving …
epoch: 1
train_loss: 0.695, train_acc: 0.499
valid_loss: 0.694, valid_acc: 0.501
Saving …
epoch: 2
train_loss: 0.693, train_acc: 0.499
valid_loss: 0.693, valid_acc: 0.501
Saving …
epoch: 3
train_loss: 0.693, train_acc: 0.496
valid_loss: 0.693, valid_acc: 0.501
Saving …
epoch: 4
train_loss: 0.694, train_acc: 0.499
valid_loss: 0.693, valid_acc: 0.501
Saving …
epoch: 5
train_loss: 0.695, train_acc: 0.503
valid_loss: 0.694, valid_acc: 0.499
test_loss: 0.694, test_acc: 0.500


hparam 43
DROPOUT_RATE=0.2,EMBEDDING_DIM=188,HIDDEN_DIM=290,LR=0.001,N_LAYERS=4,OPTIM=rmsp
rop
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 13,247,794 trainable parameters
Saving …
epoch: 1
train_loss: 0.695, train_acc: 0.499
```

```
valid_loss: 0.693, valid_acc: 0.501
Saving …
epoch: 2
train_loss: 0.694, train_acc: 0.494
valid_loss: 0.693, valid_acc: 0.499
Saving …
epoch: 3
train_loss: 0.716, train_acc: 0.503
valid_loss: 0.696, valid_acc: 0.499
Saving …
epoch: 4
train_loss: 0.705, train_acc: 0.497
valid_loss: 0.711, valid_acc: 0.504
Saving …
epoch: 5
train_loss: 0.701, train_acc: 0.503
valid_loss: 0.697, valid_acc: 0.501
test_loss: 0.698, test_acc: 0.500


hparam 44
DROPOUT_RATE=0.2,EMBEDDING_DIM=256,HIDDEN_DIM=290,LR=0.001,N_LAYERS=4,OPTIM=rmsp
rop
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 17,184,246 trainable parameters
Saving …
epoch: 1
train_loss: 0.695, train_acc: 0.502
valid_loss: 0.693, valid_acc: 0.499
Saving …
epoch: 2
train_loss: 0.693, train_acc: 0.499
valid_loss: 0.694, valid_acc: 0.499
Saving …
epoch: 3
train_loss: 0.694, train_acc: 0.499
valid_loss: 0.693, valid_acc: 0.499
Saving …
epoch: 4
train_loss: 0.694, train_acc: 0.497
valid_loss: 0.693, valid_acc: 0.499
Saving …
epoch: 5
train_loss: 0.699, train_acc: 0.500
valid_loss: 0.694, valid_acc: 0.499
```

```
test_loss: 0.693, test_acc: 0.500


hparam 45
DROPOUT_RATE=0.2,EMBEDDING_DIM=32,HIDDEN_DIM=305,LR=0.001,N_LAYERS=4,OPTIM=rmspr
op
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 4,469,440 trainable parameters
Saving …
epoch: 1
train_loss: 0.695, train_acc: 0.503
valid_loss: 0.694, valid_acc: 0.501
Saving …
epoch: 2
train_loss: 0.694, train_acc: 0.501
valid_loss: 0.697, valid_acc: 0.501
Saving …
epoch: 3
train_loss: 0.695, train_acc: 0.509
valid_loss: 0.693, valid_acc: 0.499
Saving …
epoch: 4
train_loss: 0.693, train_acc: 0.510
valid_loss: 0.694, valid_acc: 0.501
Saving …
epoch: 5
train_loss: 0.694, train_acc: 0.509
valid_loss: 0.693, valid_acc: 0.501
test_loss: 0.692, test_acc: 0.500


hparam 46
DROPOUT_RATE=0.2,EMBEDDING_DIM=188,HIDDEN_DIM=305,LR=0.001,N_LAYERS=4,OPTIM=rmsp
rop
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 13,509,484 trainable parameters
Saving …
epoch: 1
train_loss: 0.551, train_acc: 0.704
valid_loss: 0.424, valid_acc: 0.829
Saving …
epoch: 2
```

```
train_loss: 0.316, train_acc: 0.874
valid_loss: 0.384, valid_acc: 0.853
Saving …
epoch: 3
train_loss: 0.211, train_acc: 0.924
valid_loss: 0.340, valid_acc: 0.858
Saving …
epoch: 4
train_loss: 0.165, train_acc: 0.944
valid_loss: 0.331, valid_acc: 0.872
Saving …
epoch: 5
train_loss: 0.141, train_acc: 0.953
valid_loss: 0.443, valid_acc: 0.871
test_loss: 0.478, test_acc: 0.867


hparam 47
DROPOUT_RATE=0.2,EMBEDDING_DIM=256,HIDDEN_DIM=305,LR=0.001,N_LAYERS=4,OPTIM=rmsp
rop
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 17,450,016 trainable parameters
Saving …
epoch: 1
train_loss: 0.697, train_acc: 0.524
valid_loss: 0.694, valid_acc: 0.534
Saving …
epoch: 2
train_loss: 0.692, train_acc: 0.525
valid_loss: 0.692, valid_acc: 0.528
Saving …
epoch: 3
train_loss: 0.692, train_acc: 0.529
valid_loss: 0.695, valid_acc: 0.537
Saving …
epoch: 4
train_loss: 0.692, train_acc: 0.529
valid_loss: 0.691, valid_acc: 0.536
Saving …
epoch: 5
train_loss: 0.692, train_acc: 0.530
valid_loss: 0.691, valid_acc: 0.535
test_loss: 0.689, test_acc: 0.544
```

```
max_test_acc 0.8801587496485029 ; which_hp 21
diff_hp for hp that reached max_test_acc DROPOUT_RATE=0.2,EMBEDDING_DIM=32,HIDDE
N_DIM=305,LR=0.001,N_LAYERS=2,OPTIM=rmsprop
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[83], line 74
     72 hp_list_out = gen_compound_hparams()
     73 test_acc_list_compound =␣
 ↪train_compound_hparams(hp_compound_list=hp_list_out)
---> 74 diff_hparams_lst_sorted =␣
 ↪print_hparam_given_acc(hp_lst=hp_list_out, test_acc=test_acc_list_compound)

Cell In[83], line 67, in print_hparam_given_acc(hp_lst, test_acc)
     65 for idx in idx_sorted:
     66     diff_hp = diff_hparams(orig_hp=ORIG_HPARAMS, new_hp=hp_lst[idx])
---> 67     print('acc', test_acc[idx_sorted], 'diff_hparams', diff_hp)
     68     diff_hp_lst.append(diff_hp)
     69 return diff_hp_lst

TypeError: only integer scalar arrays can be converted to a scalar index
```

[87]:
```
def print_hparam_given_acc(hp_lst, test_acc):
    idx_sorted = np.argsort(test_acc)[::-1]
    diff_hp_lst = []
    for idx in idx_sorted:
        diff_hp = diff_hparams(orig_hp=ORIG_HPARAMS, new_hp=hp_lst[int(idx)])
        print('acc', test_acc[int(idx)], 'diff_hparams', diff_hp)
        diff_hp_lst.append(diff_hp)
    return diff_hp_lst


diff_hparams_lst_sorted = print_hparam_given_acc(hp_lst=hp_list_out,␣
 ↪test_acc=test_acc_list_compound)
```

```
acc 0.8801587496485029 diff_hparams DROPOUT_RATE=0.2,EMBEDDING_DIM=32,HIDDEN_DIM
=305,LR=0.001,N_LAYERS=2,OPTIM=rmsprop
acc 0.8749008133297875 diff_hparams
DROPOUT_RATE=0.2,EMBEDDING_DIM=188,HIDDEN_DIM=305,LR=0.001,OPTIM=rmsprop
acc 0.8733135115532648 diff_hparams DROPOUT_RATE=0.2,EMBEDDING_DIM=256,HIDDEN_DI
M=280,LR=0.001,N_LAYERS=2,OPTIM=rmsprop
acc 0.8722222436042059 diff_hparams
EMBEDDING_DIM=32,HIDDEN_DIM=20,LR=0.001,OPTIM=rmsprop
acc 0.8721230376334418 diff_hparams DROPOUT_RATE=0.2,EMBEDDING_DIM=32,HIDDEN_DIM
=280,LR=0.001,N_LAYERS=2,OPTIM=rmsprop
acc 0.8719246222859337 diff_hparams DROPOUT_RATE=0.2,EMBEDDING_DIM=188,HIDDEN_DI
M=290,LR=0.001,N_LAYERS=3,OPTIM=rmsprop
```

acc 0.8718254185858227 diff_hparams DROPOUT_RATE=0.2,EMBEDDING_DIM=32,HIDDEN_DIM
=20,LR=0.001,N_LAYERS=4,OPTIM=rmsprop
acc 0.8717262086414156 diff_hparams DROPOUT_RATE=0.2,EMBEDDING_DIM=256,HIDDEN_DI
M=290,LR=0.001,N_LAYERS=2,OPTIM=rmsprop
acc 0.8714285918644497 diff_hparams DROPOUT_RATE=0.2,EMBEDDING_DIM=32,HIDDEN_DIM
=290,LR=0.001,N_LAYERS=2,OPTIM=rmsprop
acc 0.8711309750874837 diff_hparams DROPOUT_RATE=0.2,EMBEDDING_DIM=188,HIDDEN_DI
M=280,LR=0.001,N_LAYERS=4,OPTIM=rmsprop
acc 0.8705357347215925 diff_hparams DROPOUT_RATE=0.2,EMBEDDING_DIM=256,HIDDEN_DI
M=280,LR=0.001,N_LAYERS=3,OPTIM=rmsprop
acc 0.8696428792817252 diff_hparams DROPOUT_RATE=0.2,EMBEDDING_DIM=188,HIDDEN_DI
M=280,LR=0.001,N_LAYERS=2,OPTIM=rmsprop
acc 0.8695436693373181 diff_hparams DROPOUT_RATE=0.2,EMBEDDING_DIM=256,HIDDEN_DI
M=20,LR=0.001,N_LAYERS=4,OPTIM=rmsprop
acc 0.8693452568281265 diff_hparams DROPOUT_RATE=0.2,EMBEDDING_DIM=188,HIDDEN_DI
M=305,LR=0.001,N_LAYERS=3,OPTIM=rmsprop
acc 0.8690476411864871 diff_hparams DROPOUT_RATE=0.2,EMBEDDING_DIM=32,HIDDEN_DIM
=20,LR=0.001,N_LAYERS=3,OPTIM=rmsprop
acc 0.8689484318097432 diff_hparams
DROPOUT_RATE=0.2,EMBEDDING_DIM=188,HIDDEN_DIM=290,LR=0.001,OPTIM=rmsprop
acc 0.8688492241359892 diff_hparams DROPOUT_RATE=0.2,EMBEDDING_DIM=32,HIDDEN_DIM
=305,LR=0.001,N_LAYERS=3,OPTIM=rmsprop
acc 0.868650814465114 diff_hparams
EMBEDDING_DIM=32,HIDDEN_DIM=20,LR=0.001,N_LAYERS=2,OPTIM=rmsprop
acc 0.8681547812053135 diff_hparams DROPOUT_RATE=0.2,EMBEDDING_DIM=188,HIDDEN_DI
M=20,LR=0.001,N_LAYERS=4,OPTIM=rmsprop
acc 0.867162717524029 diff_hparams DROPOUT_RATE=0.2,EMBEDDING_DIM=256,HIDDEN_DIM
=20,LR=0.001,N_LAYERS=2,OPTIM=rmsprop
acc 0.8671627169563657 diff_hparams DROPOUT_RATE=0.2,EMBEDDING_DIM=32,HIDDEN_DIM
=290,LR=0.001,N_LAYERS=3,OPTIM=rmsprop
acc 0.8667658947762988 diff_hparams DROPOUT_RATE=0.2,EMBEDDING_DIM=188,HIDDEN_DI
M=305,LR=0.001,N_LAYERS=4,OPTIM=rmsprop
acc 0.8661706549780709 diff_hparams DROPOUT_RATE=0.2,EMBEDDING_DIM=256,HIDDEN_DI
M=290,LR=0.001,N_LAYERS=3,OPTIM=rmsprop
acc 0.8661706538427444 diff_hparams DROPOUT_RATE=0.2,EMBEDDING_DIM=256,HIDDEN_DI
M=20,LR=0.001,N_LAYERS=3,OPTIM=rmsprop
acc 0.8650793847583589 diff_hparams DROPOUT_RATE=0.2,EMBEDDING_DIM=256,HIDDEN_DI
M=305,LR=0.001,N_LAYERS=2,OPTIM=rmsprop
acc 0.8649801804905846 diff_hparams DROPOUT_RATE=0.2,EMBEDDING_DIM=256,HIDDEN_DI
M=305,LR=0.001,N_LAYERS=3,OPTIM=rmsprop
acc 0.8645833532015482 diff_hparams
DROPOUT_RATE=0.2,EMBEDDING_DIM=256,HIDDEN_DIM=290,LR=0.001,OPTIM=rmsprop
acc 0.8635912872496105 diff_hparams DROPOUT_RATE=0.2,EMBEDDING_DIM=188,HIDDEN_DI
M=280,LR=0.001,N_LAYERS=3,OPTIM=rmsprop
acc 0.8630952590987796 diff_hparams
EMBEDDING_DIM=188,HIDDEN_DIM=20,LR=0.001,OPTIM=rmsprop
acc 0.8630952573957897 diff_hparams
DROPOUT_RATE=0.2,EMBEDDING_DIM=256,HIDDEN_DIM=280,LR=0.001,OPTIM=rmsprop

```
acc 0.8620039888790676 diff_hparams DROPOUT_RATE=0.2,EMBEDDING_DIM=188,HIDDEN_DI
M=20,LR=0.001,N_LAYERS=3,OPTIM=rmsprop
acc 0.8610119269007728 diff_hparams
DROPOUT_RATE=0.2,EMBEDDING_DIM=32,HIDDEN_DIM=305,LR=0.001,OPTIM=rmsprop
acc 0.8609127180916922 diff_hparams
DROPOUT_RATE=0.2,EMBEDDING_DIM=256,HIDDEN_DIM=305,LR=0.001,OPTIM=rmsprop
acc 0.8600198615164983 diff_hparams DROPOUT_RATE=0.2,EMBEDDING_DIM=188,HIDDEN_DI
M=290,LR=0.001,N_LAYERS=2,OPTIM=rmsprop
acc 0.8585317651430766 diff_hparams
DROPOUT_RATE=0.2,EMBEDDING_DIM=32,HIDDEN_DIM=290,LR=0.001,OPTIM=rmsprop
acc 0.8584325591723124 diff_hparams
DROPOUT_RATE=0.2,EMBEDDING_DIM=188,HIDDEN_DIM=280,LR=0.001,OPTIM=rmsprop
acc 0.857738112835657 diff_hparams
DROPOUT_RATE=0.2,EMBEDDING_DIM=256,HIDDEN_DIM=20,LR=0.001,OPTIM=rmsprop
acc 0.853373034227462 diff_hparams
EMBEDDING_DIM=188,HIDDEN_DIM=20,LR=0.001,N_LAYERS=2,OPTIM=rmsprop
acc 0.8493055758022127 diff_hparams DROPOUT_RATE=0.2,EMBEDDING_DIM=188,HIDDEN_DI
M=305,LR=0.001,N_LAYERS=2,OPTIM=rmsprop
acc 0.6358135115532648 diff_hparams
DROPOUT_RATE=0.2,EMBEDDING_DIM=32,HIDDEN_DIM=280,LR=0.001,OPTIM=rmsprop
acc 0.5437500173137302 diff_hparams DROPOUT_RATE=0.2,EMBEDDING_DIM=256,HIDDEN_DI
M=305,LR=0.001,N_LAYERS=4,OPTIM=rmsprop
acc 0.500496046316056 diff_hparams DROPOUT_RATE=0.2,EMBEDDING_DIM=32,HIDDEN_DIM=
305,LR=0.001,N_LAYERS=4,OPTIM=rmsprop
acc 0.500496046316056 diff_hparams DROPOUT_RATE=0.2,EMBEDDING_DIM=188,HIDDEN_DIM
=290,LR=0.001,N_LAYERS=4,OPTIM=rmsprop
acc 0.500496046316056 diff_hparams DROPOUT_RATE=0.2,EMBEDDING_DIM=256,HIDDEN_DIM
=280,LR=0.001,N_LAYERS=4,OPTIM=rmsprop
acc 0.49990080822081795 diff_hparams DROPOUT_RATE=0.2,EMBEDDING_DIM=256,HIDDEN_D
IM=290,LR=0.001,N_LAYERS=4,OPTIM=rmsprop
acc 0.49950398206710817 diff_hparams DROPOUT_RATE=0.2,EMBEDDING_DIM=32,HIDDEN_DI
M=290,LR=0.001,N_LAYERS=4,OPTIM=rmsprop
acc 0.49950398206710817 diff_hparams DROPOUT_RATE=0.2,EMBEDDING_DIM=32,HIDDEN_DI
M=280,LR=0.001,N_LAYERS=4,OPTIM=rmsprop
acc 0.49950398206710817 diff_hparams DROPOUT_RATE=0.2,EMBEDDING_DIM=32,HIDDEN_DI
M=280,LR=0.001,N_LAYERS=3,OPTIM=rmsprop
```

[15]:
```python
# parsing to get number of parameters and test_accuracy from text file (output␣
 ↪of one of the cells)
with open('out_lab2f.txt', 'r') as file:
    file_content = file.read()

# Regular expression pattern to split text based on "hparam idx" where idx is␣
 ↪an integer
pattern = r'(hparam \d+)'

# Find all matched patterns in the text
```

```
matches = re.finditer(pattern, file_content)

# Split the text using the pattern and include matched patterns in the output
split_strings = re.split(pattern, file_content)

# Remove empty strings from the split result
split_strings = [s.strip() for s in split_strings if s.strip()]
split_strings = [s for s in split_strings if 'hparam' not in s]

splitted_str = []
# Print the split strings along with the matched patterns
for idx, (match, string) in enumerate(zip(matches, split_strings), start=1):
    hparam_line = match.group(1)

    splitted_str.append(f'{hparam_line}\n{string}\n')

pattern = r'The model has ([\d,]+) trainable parameters.*?test_loss: (\d+\.
 ↪\d+), test_acc: (\d+\.\d+)'

idx_list = []
nparam_list = []
test_acc_list = []
# Loop through input texts and extract numbers
for i, input_text in enumerate(splitted_str):
    match = re.search(pattern, input_text, re.DOTALL)
    if match:
        number_with_commas = match.group(1)
        trainable_parameters = int(number_with_commas.replace(',', ''))
        test_loss = float(match.group(2))
        test_acc = float(match.group(3))
        print(f"Extracted data from input text {i + 1}:")
        print(f"Trainable Parameters: {trainable_parameters}")
        print(f"Test Loss: {test_loss}")
        print(f"Test Accuracy: {test_acc}\n")
        test_acc_list.append(test_acc)
        nparam_list.append(trainable_parameters)
        idx_list.append(i)
    else:
        print(f"Data not found in input text {i + 1}.")

idx_sort_test_acc = np.argsort(np.asarray(test_acc_list))[::-1]
sorted_nparam_list = np.asarray(nparam_list)[idx_sort_test_acc]
plt.scatter(np.asarray(test_acc_list)[idx_sort_test_acc], sorted_nparam_list,␣
 ↪s=3)
plt.xlabel('Test Accuracy')
plt.ylabel('Number of Trainable Parameters')
plt.title('Number of trainable Parameters vs Test Accuracy (lab2f)')
```

```
# plt.savefig('q2f_num_param_vs_acc.pdf', dpi=500, bbox_inches='tight')
```

```
Extracted data from input text 1:
Trainable Parameters: 1819690
Test Loss: 0.579
Test Accuracy: 0.872

Extracted data from input text 2:
Trainable Parameters: 10681894
Test Loss: 0.632
Test Accuracy: 0.863

Extracted data from input text 3:
Trainable Parameters: 14544906
Test Loss: 0.57
Test Accuracy: 0.858

Extracted data from input text 4:
Trainable Parameters: 2167570
Test Loss: 0.692
Test Accuracy: 0.636

Extracted data from input text 5:
Trainable Parameters: 11192014
Test Loss: 0.652
Test Accuracy: 0.858

Extracted data from input text 6:
Trainable Parameters: 15125746
Test Loss: 0.617
Test Accuracy: 0.863

Extracted data from input text 7:
Trainable Parameters: 2191750
Test Loss: 0.364
Test Accuracy: 0.859

Extracted data from input text 8:
Trainable Parameters: 11222434
Test Loss: 0.666
Test Accuracy: 0.869

Extracted data from input text 9:
Trainable Parameters: 15158886
Test Loss: 0.6
Test Accuracy: 0.865
```

```
Extracted data from input text 10:
Trainable Parameters: 2229520
Test Loss: 0.701
Test Accuracy: 0.861

Extracted data from input text 11:
Trainable Parameters: 11269564
Test Loss: 0.61
Test Accuracy: 0.875

Extracted data from input text 12:
Trainable Parameters: 15210096
Test Loss: 0.596
Test Accuracy: 0.861

Extracted data from input text 13:
Trainable Parameters: 1823050
Test Loss: 0.492
Test Accuracy: 0.869

Extracted data from input text 14:
Trainable Parameters: 10685254
Test Loss: 0.558
Test Accuracy: 0.853

Extracted data from input text 15:
Trainable Parameters: 14548266
Test Loss: 0.586
Test Accuracy: 0.867

Extracted data from input text 16:
Trainable Parameters: 2797010
Test Loss: 0.39
Test Accuracy: 0.872

Extracted data from input text 17:
Trainable Parameters: 11821454
Test Loss: 0.621
Test Accuracy: 0.87

Extracted data from input text 18:
Trainable Parameters: 15755186
Test Loss: 0.449
Test Accuracy: 0.873

Extracted data from input text 19:
Trainable Parameters: 2866870
Test Loss: 0.442
```

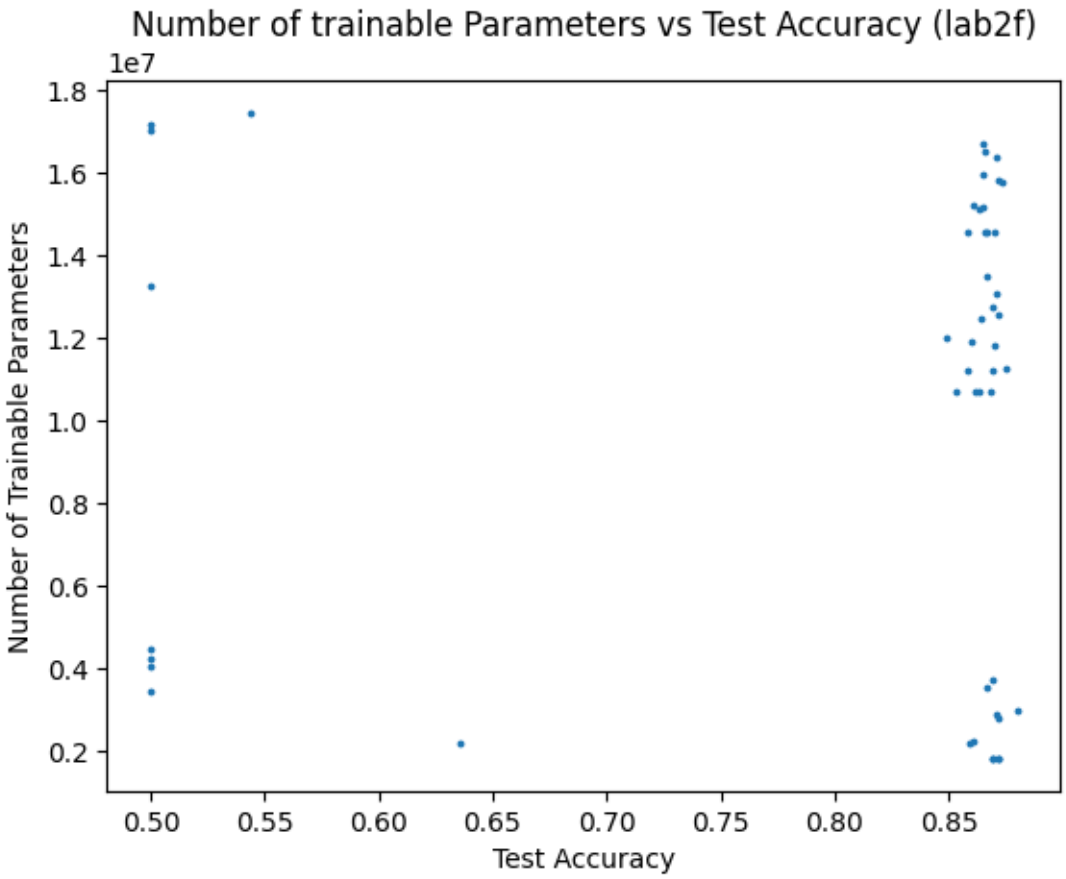
```
Test Accuracy: 0.871

Extracted data from input text 20:
Trainable Parameters: 11897554
Test Loss: 0.421
Test Accuracy: 0.86

Extracted data from input text 21:
Trainable Parameters: 15834006
Test Loss: 0.455
Test Accuracy: 0.872

Extracted data from input text 22:
Trainable Parameters: 2976160
Test Loss: 0.441
Test Accuracy: 0.88

Extracted data from input text 23:
Trainable Parameters: 12016204
Test Loss: 0.573
Test Accuracy: 0.849

Extracted data from input text 24:
Trainable Parameters: 15956736
Test Loss: 0.389
Test Accuracy: 0.865

Extracted data from input text 25:
Trainable Parameters: 1826410
Test Loss: 0.448
Test Accuracy: 0.869

Extracted data from input text 26:
Trainable Parameters: 10688614
Test Loss: 0.447
Test Accuracy: 0.862

Extracted data from input text 27:
Trainable Parameters: 14551626
Test Loss: 0.437
Test Accuracy: 0.866

Extracted data from input text 28:
Trainable Parameters: 3426450
Test Loss: 0.69
Test Accuracy: 0.5

Extracted data from input text 29:
```

```
Trainable Parameters: 12450894
Test Loss: 0.399
Test Accuracy: 0.864

Extracted data from input text 30:
Trainable Parameters: 16384626
Test Loss: 0.374
Test Accuracy: 0.871

Extracted data from input text 31:
Trainable Parameters: 3541990
Test Loss: 0.35
Test Accuracy: 0.867

Extracted data from input text 32:
Trainable Parameters: 12572674
Test Loss: 0.362
Test Accuracy: 0.872

Extracted data from input text 33:
Trainable Parameters: 16509126
Test Loss: 0.419
Test Accuracy: 0.866

Extracted data from input text 34:
Trainable Parameters: 3722800
Test Loss: 0.317
Test Accuracy: 0.869

Extracted data from input text 35:
Trainable Parameters: 12762844
Test Loss: 0.59
Test Accuracy: 0.869

Extracted data from input text 36:
Trainable Parameters: 16703376
Test Loss: 0.449
Test Accuracy: 0.865

Extracted data from input text 37:
Trainable Parameters: 1829770
Test Loss: 0.546
Test Accuracy: 0.872

Extracted data from input text 38:
Trainable Parameters: 10691974
Test Loss: 0.41
Test Accuracy: 0.868
```

```
Extracted data from input text 39:
Trainable Parameters: 14554986
Test Loss: 0.507
Test Accuracy: 0.87

Extracted data from input text 40:
Trainable Parameters: 4055890
Test Loss: 0.694
Test Accuracy: 0.5

Extracted data from input text 41:
Trainable Parameters: 13080334
Test Loss: 0.444
Test Accuracy: 0.871

Extracted data from input text 42:
Trainable Parameters: 17014066
Test Loss: 0.693
Test Accuracy: 0.5

Extracted data from input text 43:
Trainable Parameters: 4217110
Test Loss: 0.694
Test Accuracy: 0.5

Extracted data from input text 44:
Trainable Parameters: 13247794
Test Loss: 0.698
Test Accuracy: 0.5

Extracted data from input text 45:
Trainable Parameters: 17184246
Test Loss: 0.693
Test Accuracy: 0.5

Extracted data from input text 46:
Trainable Parameters: 4469440
Test Loss: 0.692
Test Accuracy: 0.5

Extracted data from input text 47:
Trainable Parameters: 13509484
Test Loss: 0.478
Test Accuracy: 0.867

Extracted data from input text 48:
Trainable Parameters: 17450016
```

```
Test Loss: 0.689
Test Accuracy: 0.544
```

[15]:
```
Text(0.5, 1.0, 'Number of trainable Parameters vs Test Accuracy (lab2f)')
```



[22]:
```
input_t = splitted_str

trainable_params_lst = []
test_acc_lst = []
for input_text in input_t:
    trainable_parameters = re.search(r'The model has (\d+(,\d{3})*) trainable␣
 ↪parameters', input_text)
    test_loss_acc = re.search(r'test_loss: (\d+\.\d+), test_acc: (\d+\.\d+)',␣
 ↪input_text)

    # Extracted values
    trainable_parameters_value = trainable_parameters.group(1).replace(',', '')
    test_loss_value = test_loss_acc.group(1)
```

```
    test_acc_value = test_loss_acc.group(2)

    trainable_params_lst.append(int(trainable_parameters_value))
    test_acc_lst.append(float(test_acc_value))
    # Print the extracted values
    # print("Trainable Parameters:", trainable_parameters_value)
    # print("Test Loss:", test_loss_value)
    # print("Test Accuracy:", test_acc_value)

def rank_given_acc_and_num_params(test_acc=test_acc_lst,␣
 ↪num_params=trainable_params_lst):
    ratio = [test_acc[i] / num_params[i] for i in range(len(test_acc_lst))]
    idx_sorted = np.argsort(np.array(ratio))[::-1]
    for idx in idx_sorted:
        print('index %d, acc %.5f, num_params %d, acc/num_params %g'
              % (idx, test_acc[int(idx)], num_params[int(idx)],␣
 ↪test_acc[int(idx)] / num_params[int(idx)]))
    return

rank_given_acc_and_num_params()
# note: for "acc %.5f" e.g. "acc 0.87200" in the first line, disregard trailing␣
 ↪zeros. The precision is to
# 3 decimal places only.
```

```
index 0, acc 0.87200, num_params 1819690, acc/num_params 4.79203e-07
index 12, acc 0.86900, num_params 1823050, acc/num_params 4.76674e-07
index 36, acc 0.87200, num_params 1829770, acc/num_params 4.76563e-07
index 24, acc 0.86900, num_params 1826410, acc/num_params 4.75797e-07
index 6, acc 0.85900, num_params 2191750, acc/num_params 3.91924e-07
index 9, acc 0.86100, num_params 2229520, acc/num_params 3.86182e-07
index 15, acc 0.87200, num_params 2797010, acc/num_params 3.11761e-07
index 18, acc 0.87100, num_params 2866870, acc/num_params 3.03816e-07
index 21, acc 0.88000, num_params 2976160, acc/num_params 2.95683e-07
index 3, acc 0.63600, num_params 2167570, acc/num_params 2.93416e-07
index 30, acc 0.86700, num_params 3541990, acc/num_params 2.44778e-07
index 33, acc 0.86900, num_params 3722800, acc/num_params 2.33426e-07
index 27, acc 0.50000, num_params 3426450, acc/num_params 1.45924e-07
index 39, acc 0.50000, num_params 4055890, acc/num_params 1.23278e-07
index 42, acc 0.50000, num_params 4217110, acc/num_params 1.18565e-07
index 45, acc 0.50000, num_params 4469440, acc/num_params 1.11871e-07
index 37, acc 0.86800, num_params 10691974, acc/num_params 8.11824e-08
index 1, acc 0.86300, num_params 10681894, acc/num_params 8.07909e-08
index 25, acc 0.86200, num_params 10688614, acc/num_params 8.06466e-08
index 13, acc 0.85300, num_params 10685254, acc/num_params 7.98296e-08
index 10, acc 0.87500, num_params 11269564, acc/num_params 7.76428e-08
index 7, acc 0.86900, num_params 11222434, acc/num_params 7.74342e-08
index 4, acc 0.85800, num_params 11192014, acc/num_params 7.66618e-08
```

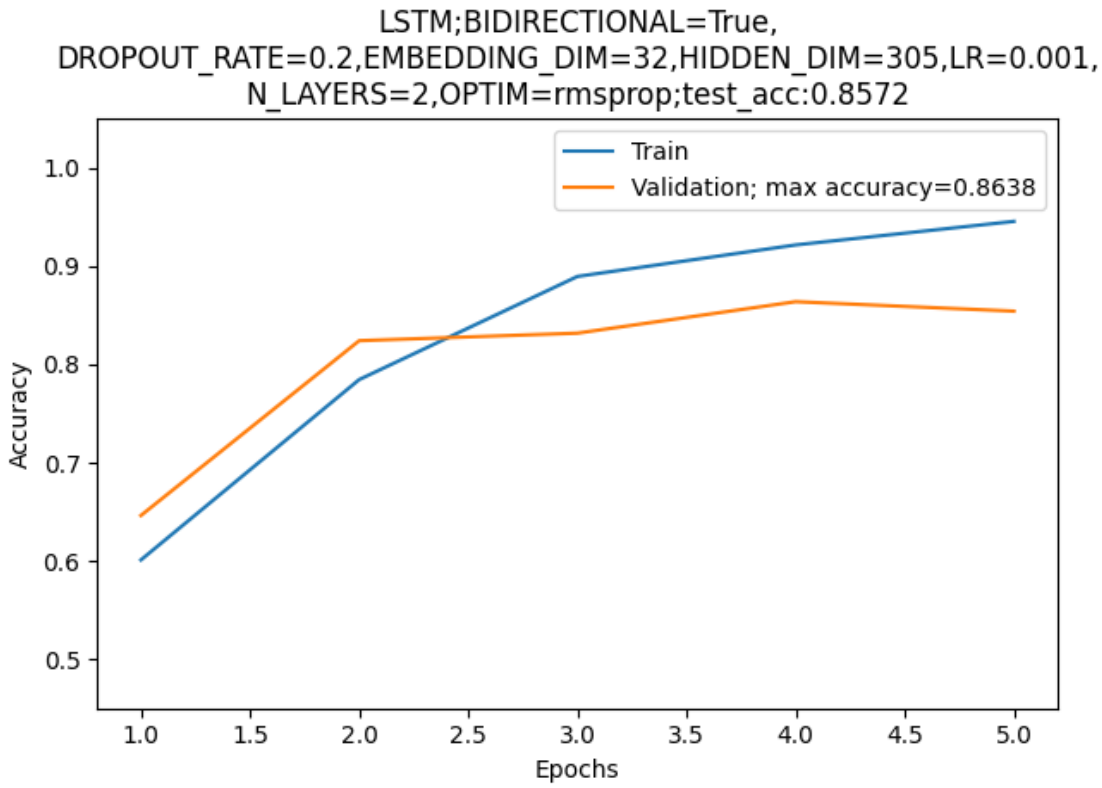
```
index 16, acc 0.87000, num_params 11821454, acc/num_params 7.3595e-08
index 19, acc 0.86000, num_params 11897554, acc/num_params 7.22838e-08
index 22, acc 0.84900, num_params 12016204, acc/num_params 7.06546e-08
index 28, acc 0.86400, num_params 12450894, acc/num_params 6.93926e-08
index 31, acc 0.87200, num_params 12572674, acc/num_params 6.93568e-08
index 34, acc 0.86900, num_params 12762844, acc/num_params 6.80883e-08
index 40, acc 0.87100, num_params 13080334, acc/num_params 6.65885e-08
index 46, acc 0.86700, num_params 13509484, acc/num_params 6.41771e-08
index 38, acc 0.87000, num_params 14554986, acc/num_params 5.97733e-08
index 14, acc 0.86700, num_params 14548266, acc/num_params 5.95947e-08
index 26, acc 0.86600, num_params 14551626, acc/num_params 5.95122e-08
index 2, acc 0.85800, num_params 14544906, acc/num_params 5.89897e-08
index 8, acc 0.86500, num_params 15158886, acc/num_params 5.70622e-08
index 5, acc 0.86300, num_params 15125746, acc/num_params 5.7055e-08
index 11, acc 0.86100, num_params 15210096, acc/num_params 5.66071e-08
index 17, acc 0.87300, num_params 15755186, acc/num_params 5.54103e-08
index 20, acc 0.87200, num_params 15834006, acc/num_params 5.50713e-08
index 23, acc 0.86500, num_params 15956736, acc/num_params 5.42091e-08
index 29, acc 0.87100, num_params 16384626, acc/num_params 5.31596e-08
index 32, acc 0.86600, num_params 16509126, acc/num_params 5.24558e-08
index 35, acc 0.86500, num_params 16703376, acc/num_params 5.17859e-08
index 43, acc 0.50000, num_params 13247794, acc/num_params 3.77421e-08
index 47, acc 0.54400, num_params 17450016, acc/num_params 3.11748e-08
index 41, acc 0.50000, num_params 17014066, acc/num_params 2.93874e-08
index 44, acc 0.50000, num_params 17184246, acc/num_params 2.90964e-08
```

1.6.12 Lab 2 (g) Bi-Directional LSTM, using best architecture from (f)

[89]:
```
# diff_hparams DROPOUT_RATE=0.2,EMBEDDING_DIM=32,HIDDEN_DIM=305,LR=0.
 ↪001,N_LAYERS=2,OPTIM=rmsprop
# note: this is the optimal hparam setting for BIDIRECTIONAL = False when␣
 ↪considering accuracy only
bi_dir_hp = HyperParams()

bi_dir_hp.BIDIRECTIONAL = True

bi_dir_hp.DROPOUT_RATE = 0.2
bi_dir_hp.EMBEDDING_DIM = 32
bi_dir_hp.HIDDEN_DIM = 305
bi_dir_hp.LR = 0.001
bi_dir_hp.N_LAYERS = 2
bi_dir_hp.OPTIM = 'rmsprop'

print(diff_hparams(orig_hp=ORIG_HPARAMS, new_hp=bi_dir_hp))

rd_2g_bidirect = train_and_test_model_with_hparams(hparams=bi_dir_hp,␣
 ↪model_type="lstm")
```

```
BIDIRECTIONAL=True,DROPOUT_RATE=0.2,EMBEDDING_DIM=32,HIDDEN_DIM=305,LR=0.001,N_L
AYERS=2,OPTIM=rmsprop
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 4,881,190 trainable parameters
Saving …
epoch: 1
train_loss: 0.669, train_acc: 0.601
valid_loss: 0.613, valid_acc: 0.646
Saving …
epoch: 2
train_loss: 0.463, train_acc: 0.785
valid_loss: 0.399, valid_acc: 0.824
Saving …
epoch: 3
train_loss: 0.275, train_acc: 0.889
valid_loss: 0.384, valid_acc: 0.832
Saving …
epoch: 4
train_loss: 0.205, train_acc: 0.922
valid_loss: 0.386, valid_acc: 0.864
Saving …
epoch: 5
train_loss: 0.152, train_acc: 0.946
valid_loss: 0.433, valid_acc: 0.854
test_loss: 0.429, test_acc: 0.857
```

[124]:
```
plot_train_val_acc(save_name='q2g_bestHP', ret_dict=rd_2g_bidirect,␣
↪orig_hp=ORIG_HPARAMS,
                   new_hp=bi_dir_hp, save='y', gru=False)
```

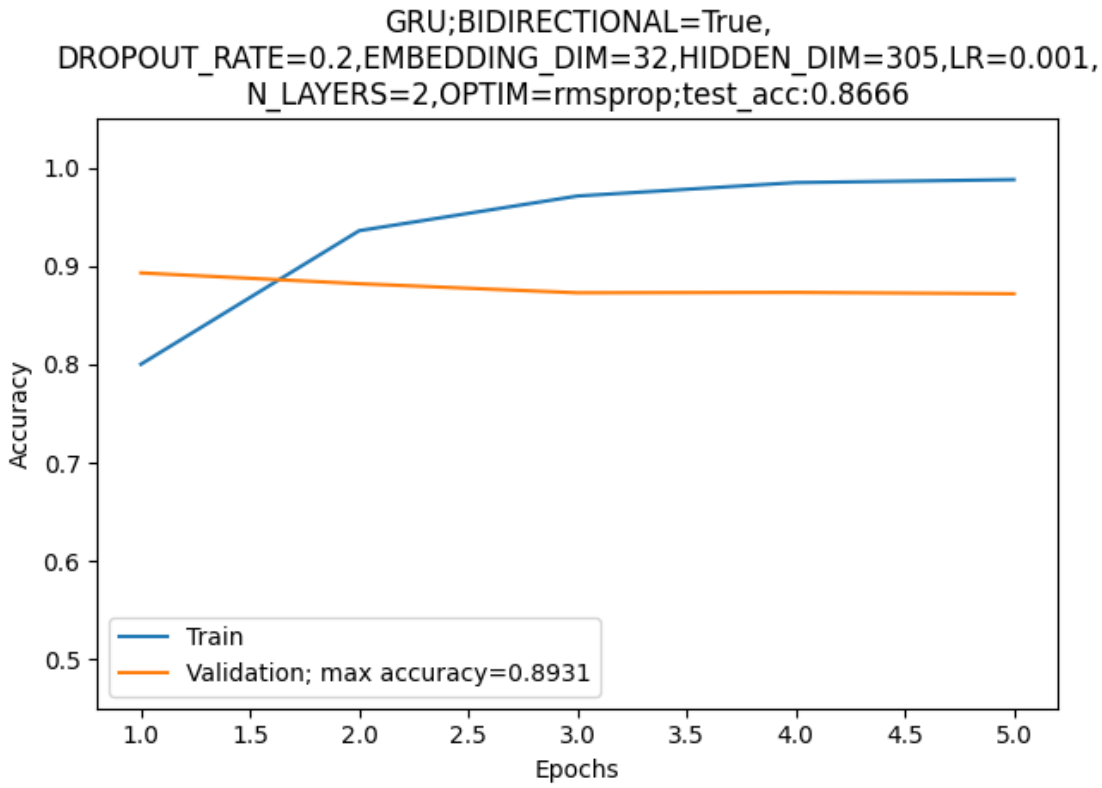


[14]:
```
# diff_hparams DROPOUT_RATE=0.2,EMBEDDING_DIM=32,HIDDEN_DIM=305,LR=0.
 ↪001,N_LAYERS=2,OPTIM=rmsprop
# note: this is the optimal hparam setting for BIDIRECTIONAL = False
# (found using LSTM when considering accuracy only)
# this is trained using the GRU model
bi_dir_hp_gru = HyperParams()

bi_dir_hp_gru.BIDIRECTIONAL = True

bi_dir_hp_gru.DROPOUT_RATE = 0.2
bi_dir_hp_gru.EMBEDDING_DIM = 32
bi_dir_hp_gru.HIDDEN_DIM = 305
bi_dir_hp_gru.LR = 0.001
bi_dir_hp_gru.N_LAYERS = 2
bi_dir_hp_gru.OPTIM = 'rmsprop'

print(diff_hparams(orig_hp=ORIG_HPARAMS, new_hp=bi_dir_hp_gru))

rd_2g_bidirect_gru = train_and_test_model_with_hparams(hparams=bi_dir_hp_gru,
                                                       model_type="gru",
```

```
                                                                    ␣
↪override_models_with_gru=True)
```

```
BIDIRECTIONAL=True,
DROPOUT_RATE=0.2,EMBEDDING_DIM=32,HIDDEN_DIM=305,LR=0.001,
N_LAYERS=2,OPTIM=rmsprop
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 4,115,030 trainable parameters
Saving …
epoch: 1
train_loss: 0.425, train_acc: 0.800
valid_loss: 0.283, valid_acc: 0.893
Saving …
epoch: 2
train_loss: 0.185, train_acc: 0.936
valid_loss: 0.349, valid_acc: 0.882
Saving …
epoch: 3
train_loss: 0.085, train_acc: 0.971
valid_loss: 0.545, valid_acc: 0.873
Saving …
epoch: 4
train_loss: 0.047, train_acc: 0.985
valid_loss: 0.541, valid_acc: 0.873
Saving …
epoch: 5
train_loss: 0.041, train_acc: 0.988
valid_loss: 0.910, valid_acc: 0.872
test_loss: 0.924, test_acc: 0.867
```

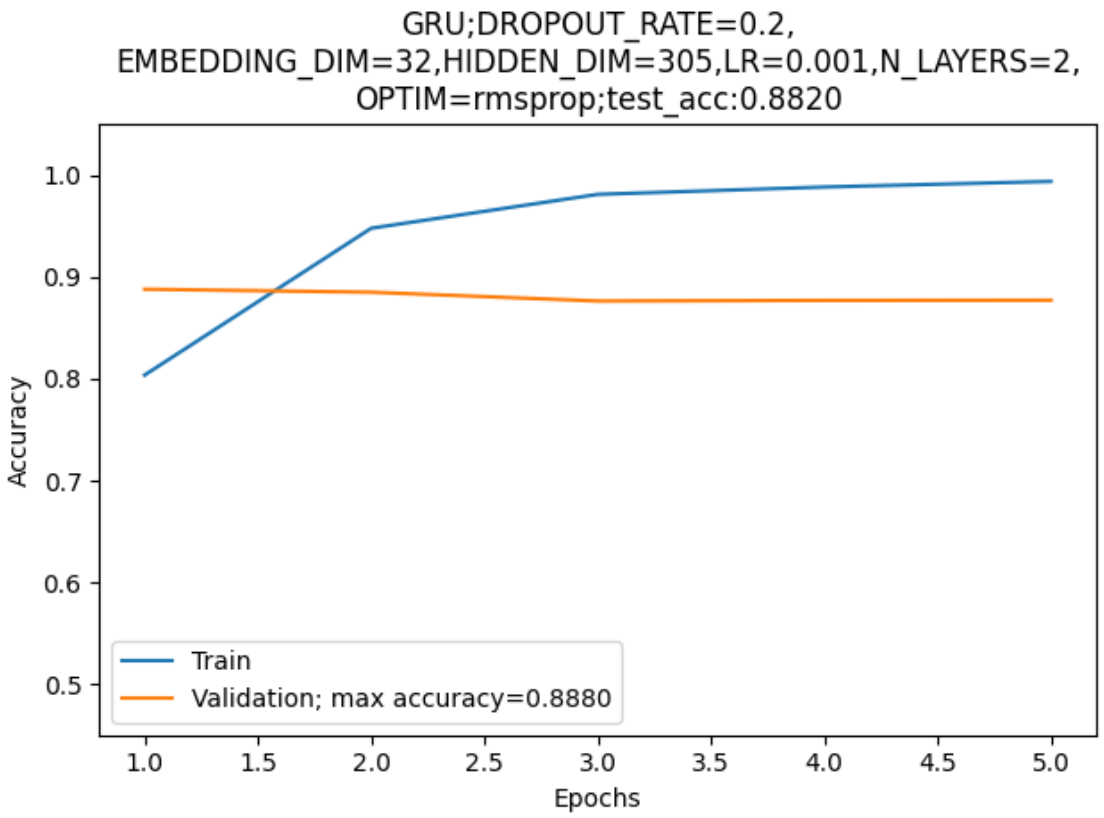
[22]:
```
plot_train_val_acc(save_name='q2g_bestHP_gru_bidir',␣
↪ret_dict=rd_2g_bidirect_gru, orig_hp=ORIG_HPARAMS,
                   new_hp=bi_dir_hp_gru, save='y', gru=True)
```



[19]:
```
# diff_hparams DROPOUT_RATE=0.2,EMBEDDING_DIM=32,HIDDEN_DIM=305,LR=0.
 ↪001,N_LAYERS=2,OPTIM=rmsprop
# note: this is the optimal hparam setting for BIDIRECTIONAL = False (found␣
 ↪using LSTM)
best_hp_gru = HyperParams()

best_hp_gru.BIDIRECTIONAL = False

best_hp_gru.DROPOUT_RATE = 0.2
best_hp_gru.EMBEDDING_DIM = 32
best_hp_gru.HIDDEN_DIM = 305
best_hp_gru.LR = 0.001
best_hp_gru.N_LAYERS = 2
best_hp_gru.OPTIM = 'rmsprop'

print(diff_hparams(orig_hp=ORIG_HPARAMS, new_hp=best_hp_gru))

rd_2g_best_hp_gru = train_and_test_model_with_hparams(hparams=best_hp_gru,
                                                      model_type="gru",
                                                     ␣
 ↪override_models_with_gru=True)
```

```
DROPOUT_RATE=0.2,
EMBEDDING_DIM=32,HIDDEN_DIM=305,LR=0.001,N_LAYERS=2,
OPTIM=rmsprop
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 2,686,105 trainable parameters
Saving …
epoch: 1
train_loss: 0.421, train_acc: 0.804
valid_loss: 0.268, valid_acc: 0.888
Saving …
epoch: 2
train_loss: 0.144, train_acc: 0.948
valid_loss: 0.318, valid_acc: 0.885
Saving …
epoch: 3
train_loss: 0.056, train_acc: 0.981
valid_loss: 0.379, valid_acc: 0.876
Saving …
epoch: 4
train_loss: 0.033, train_acc: 0.988
valid_loss: 0.541, valid_acc: 0.877
Saving …
epoch: 5
train_loss: 0.019, train_acc: 0.994
valid_loss: 0.654, valid_acc: 0.877
test_loss: 0.635, test_acc: 0.882
```

[20]:
```
plot_train_val_acc(save_name='q2g_bestHP_gru_singledir',␣
  ↪ret_dict=rd_2g_best_hp_gru, orig_hp=ORIG_HPARAMS,
                    new_hp=best_hp_gru, save='y', gru=True)
```

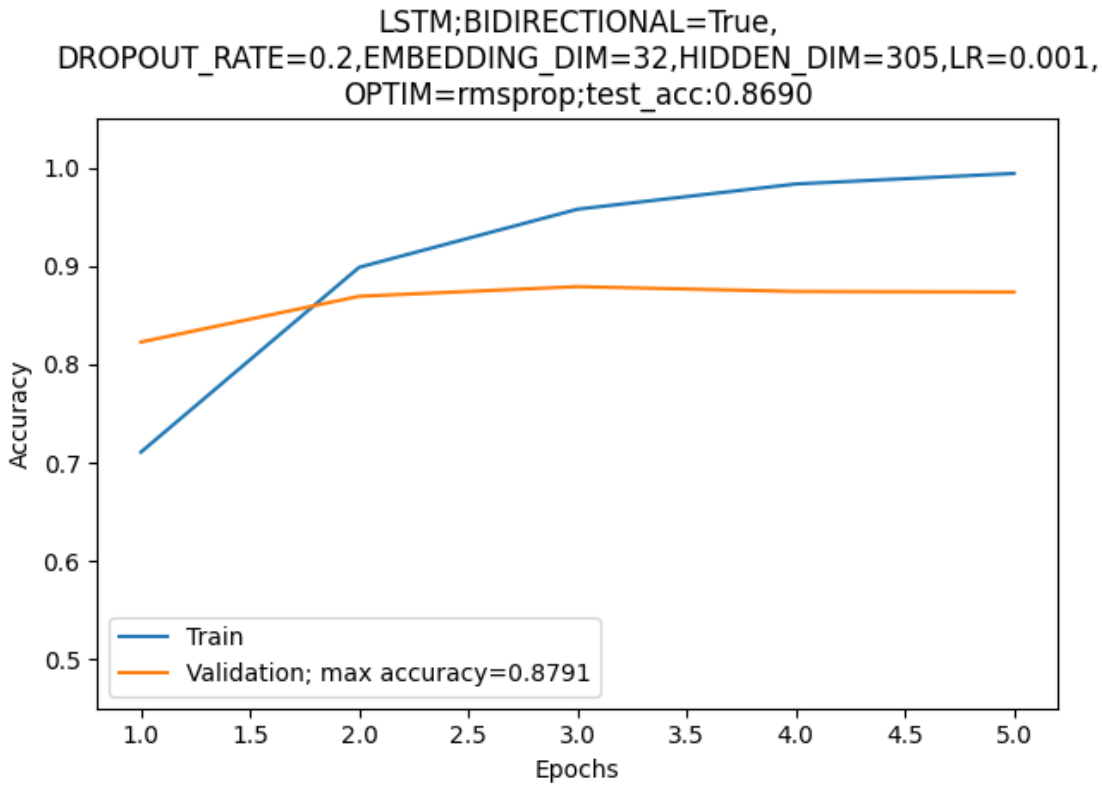


[96]:
```
# diff_hparams DROPOUT_RATE=0.2,EMBEDDING_DIM=32,HIDDEN_DIM=305,LR=0.
 ↪001,N_LAYERS=1,OPTIM=rmsprop
# note: this is the optimal hparam setting for BIDIRECTIONAL = False EXCEPT␣
 ↪N_LAYERS=1
# in this case
# this means that the model has less trainable parameters

bi_dir_hp2 = HyperParams()

bi_dir_hp2.BIDIRECTIONAL = True

bi_dir_hp2.DROPOUT_RATE = 0.2
bi_dir_hp2.EMBEDDING_DIM = 32
bi_dir_hp2.HIDDEN_DIM = 305
bi_dir_hp2.LR = 0.001
bi_dir_hp2.N_LAYERS = 1
bi_dir_hp2.OPTIM = 'rmsprop'

print(diff_hparams(orig_hp=ORIG_HPARAMS, new_hp=bi_dir_hp2))
```

```
rd_2g2_bidirect = train_and_test_model_with_hparams(hparams=bi_dir_hp2,␣
↪model_type="lstm")
```

```
BIDIRECTIONAL=True,
DROPOUT_RATE=0.2,EMBEDDING_DIM=32,HIDDEN_DIM=305,LR=0.001,
OPTIM=rmsprop
shape of train data is 35000
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 2,643,710 trainable parameters
Saving …
epoch: 1
train_loss: 0.566, train_acc: 0.711
valid_loss: 0.394, valid_acc: 0.823
Saving …
epoch: 2
train_loss: 0.253, train_acc: 0.899
valid_loss: 0.313, valid_acc: 0.869
Saving …
epoch: 3
train_loss: 0.118, train_acc: 0.958
valid_loss: 0.338, valid_acc: 0.879
Saving …
epoch: 4
train_loss: 0.049, train_acc: 0.984
valid_loss: 0.429, valid_acc: 0.874
Saving …
epoch: 5
train_loss: 0.018, train_acc: 0.994
valid_loss: 0.579, valid_acc: 0.874
test_loss: 0.585, test_acc: 0.869
```

[97]:
```
plot_train_val_acc(save_name='q2g_otherHP', ret_dict=rd_2g2_bidirect,␣
↪orig_hp=ORIG_HPARAMS,
                   new_hp=bi_dir_hp2, save='y', gru=False)
```

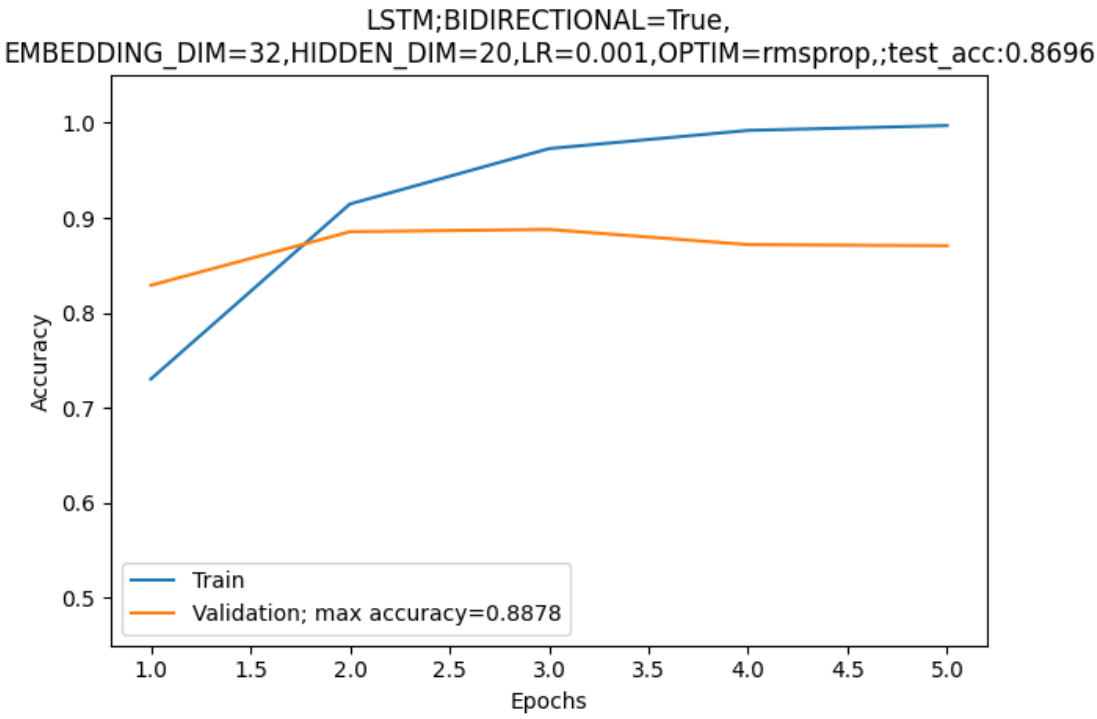


[23]:
```
# hparam 0: EMBEDDING_DIM=32,HIDDEN_DIM=20,LR=0.001,OPTIM=rmsprop
# this is the best model using acc/num_param criterion

bi_dir_hp_acc_num_param = HyperParams()

bi_dir_hp_acc_num_param.BIDIRECTIONAL = True

bi_dir_hp_acc_num_param.EMBEDDING_DIM = 32
bi_dir_hp_acc_num_param.HIDDEN_DIM = 20
bi_dir_hp_acc_num_param.LR = 0.001
bi_dir_hp_acc_num_param.OPTIM = 'rmsprop'

print(diff_hparams(orig_hp=ORIG_HPARAMS, new_hp=bi_dir_hp_acc_num_param))

rd_2g_bidirect_acc_numparam =␣
 ↪train_and_test_model_with_hparams(hparams=bi_dir_hp_acc_num_param,
                                                                  ␣
 ↪model_type="lstm")
```

```
BIDIRECTIONAL=True,
EMBEDDING_DIM=32,HIDDEN_DIM=20,LR=0.001,OPTIM=rmsprop,
shape of train data is 35000
```

```
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 1,824,050 trainable parameters
Saving …
epoch: 1
train_loss: 0.533, train_acc: 0.730
valid_loss: 0.401, valid_acc: 0.829
Saving …
epoch: 2
train_loss: 0.220, train_acc: 0.915
valid_loss: 0.289, valid_acc: 0.885
Saving …
epoch: 3
train_loss: 0.084, train_acc: 0.973
valid_loss: 0.352, valid_acc: 0.888
Saving …
epoch: 4
train_loss: 0.028, train_acc: 0.992
valid_loss: 0.491, valid_acc: 0.872
Saving …
epoch: 5
train_loss: 0.011, train_acc: 0.997
valid_loss: 0.646, valid_acc: 0.871
test_loss: 0.638, test_acc: 0.870
```

[28]:
```
plot_train_val_acc(save_name='q2g_bidir_bestHP_acc_nparams',␣
↪ret_dict=rd_2g_bidirect_acc_numparam,
                   orig_hp=ORIG_HPARAMS, new_hp=bi_dir_hp_acc_num_param,␣
↪save='y', gru=False)
```

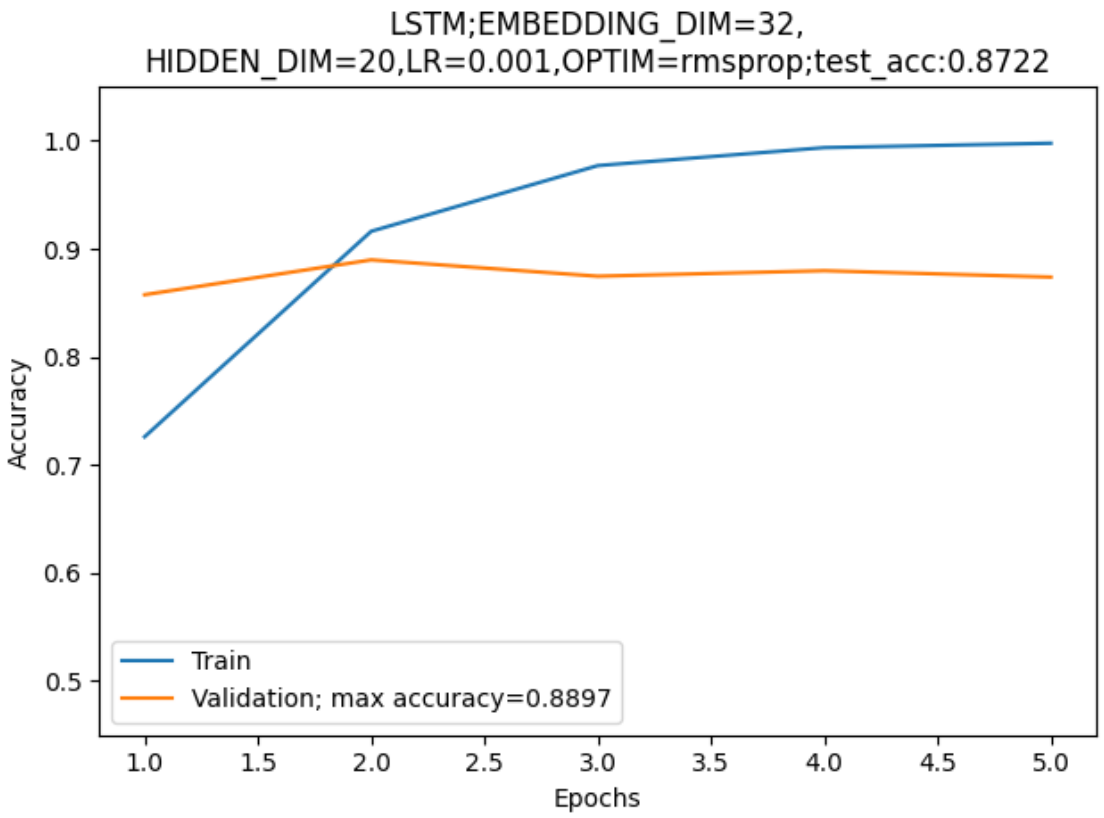


[25]:
```
# hparam 0: EMBEDDING_DIM=32,HIDDEN_DIM=20,LR=0.001,OPTIM=rmsprop
# this is the best model using acc/num_param criterion (found using LSTM)

## NOTE: THIS IS FOR UNI-DIRECTIONAL, as a comparison

best_hp_acc_num_param = HyperParams()

best_hp_acc_num_param.BIDIRECTIONAL = False

best_hp_acc_num_param.EMBEDDING_DIM = 32
best_hp_acc_num_param.HIDDEN_DIM = 20
best_hp_acc_num_param.LR = 0.001
best_hp_acc_num_param.OPTIM = 'rmsprop'

print(diff_hparams(orig_hp=ORIG_HPARAMS, new_hp=best_hp_acc_num_param))

rd_2g_best_acc_numparam =␣
 ↪train_and_test_model_with_hparams(hparams=best_hp_acc_num_param,
                                                                 ␣
 ↪model_type="lstm")
```

```
EMBEDDING_DIM=32,
HIDDEN_DIM=20,LR=0.001,OPTIM=rmsprop
shape of train data is 35000
```

```
shape of test data is 10000
shape of valid data is 5000
Length of vocabulary is 56729
The model has 1,819,690 trainable parameters
Saving …
epoch: 1
train_loss: 0.549, train_acc: 0.726
valid_loss: 0.335, valid_acc: 0.858
Saving …
epoch: 2
train_loss: 0.216, train_acc: 0.916
valid_loss: 0.278, valid_acc: 0.890
Saving …
epoch: 3
train_loss: 0.071, train_acc: 0.977
valid_loss: 0.372, valid_acc: 0.875
Saving …
epoch: 4
train_loss: 0.022, train_acc: 0.994
valid_loss: 0.482, valid_acc: 0.880
Saving …
epoch: 5
train_loss: 0.008, train_acc: 0.998
valid_loss: 0.559, valid_acc: 0.874
test_loss: 0.579, test_acc: 0.872
```

[27]:
```
plot_train_val_acc(save_name='q2g_unidir_bestHP_acc_nparams',␣
↪ret_dict=rd_2g_best_acc_numparam,
                   orig_hp=ORIG_HPARAMS, new_hp=best_hp_acc_num_param,␣
↪save='y', gru=False)
```


LSTM;EMBEDDING_DIM=32,
HIDDEN_DIM=20,LR=0.001,OPTIM=rmsprop;test_acc:0.8722
Accuracy
Epochs
Train
Validation; max accuracy=0.8897
1.0
0.9
0.8
0.7
0.6
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0

[ ]: