CS 512: Advanced Machine Learning

Lab 2: Conditional Random Fields with Convolutions

Student: Amrit Raj Vardhan *Email: avardh5@uic.edu*

Harsh Mishra (hmishr3@uic.edu), Amrit Raj Vardhan (avardh5@uic.edu), Karan Jogi (kjogi2@uic.edu),

Manmohan Dogra (mdogra3@uic.edu), Nikita Thakur (nthaku3@uic.edu)

Deadline: 4 PM, March 28, 2022

In this project we will continue to explore Conditional Random Fields (CRFs), but we will use additional image level features such as aconvolutions to aid the training. We will use PyTorch to implement our CRF model and convolutions. You will get a chance to implement an end-to-end machine learning solution to a problem in PyTorch. In the process, you will also pick up tools to do differentiable layer-wise programming, which is common across all the popular deep learning frameworks that exist today.

Acceptable libraries. The main benefits of using PyTorch (and other similar deep learning libraries) is GPU computation and features such as automatic differentiation. Although you could technically use libraries such as numpy for performing certain numerical computations inside your layer, you may lose these benefits in doing so.

How to submit. Only one member of each team needs to submit a zip file on Gradescope under Lab 1. The filename should be `Firstname_Lastname.zip`, where both names correspond to the member who submits it. Make sure you also indicate your **teammates** by selecting on Gradescope.

Inside the zip/tar file, the following contents should be included:

1. A PDF report named `Report_Firstname_Lastname.pdf` with answers to the questions outlined below. **Your report should include the name and NetID of *all* team members.** The LaTeX source code of this document is provided with the package, and you may write up your report based on it.
2. Your source code, which should be well commented. Include a short `readme.txt` file that explains how to run your code.

You are allowed to resubmit as often as you like and your grade will be based on the last version submitted. Late submissions will not be accepted in any case, unless there is a documented personal emergency. Arrangements must be made with the instructor as soon as possible after the emergency arises, preferably well before the deadline. This assignment contributes **11%** to your final grade.

1 Introduction

Dataset. We will use Taskar's OCR dataset for this project(the same dataset which was used in Assignment2). The dataset description is repeated here again for your convenience.

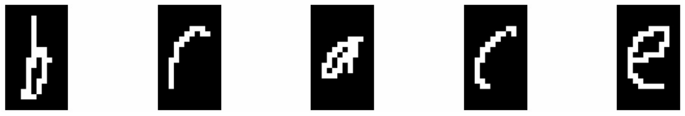


Figure 1. Example word image

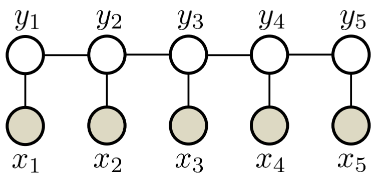


Figure 2. CRF for word-letter

The original dataset was maintained by Ben Taskar. It contains the image and label of 6,877 words collected from 150 human subjects, with 52,152 letters in total. To simplify feature engineering, each letter image is encoded by a 128 (=16*8) dimensional vector, whose entries are either 0 (black) or 1 (white). Note in this dataset, only lowercase letters are involved, *i.e.* 26 possible labels. Since the first letter of each word was capitalized and the rest were in lowercase, the dataset has removed all first letters.

Minibatches. When training a model in PyTorch, it is common to send minibatches of training examples to the optimizer. A minibatch is a small subset of the training data (the size of the minibatch is usually a tunable hyperparameter).

In the starter code that is provided, the `DataLoader` is already setup to process the input dataset as batches. The entire dataset is also divided evenly into training and test data in `train.py`.

However, when implementing the CRF and Convolution layer, you will have to be mindful of the fact that the input to your module will be a minibatch. The following is the shape of the data after batching.

$$X \in \mathbb{R}^{\text{batch\_size} \times \text{max\_word\_length} \times 128} \tag{1}$$

So every row of a batch corresponds to a word (sequence). Here max_word_length is the maximum word length of all the words in the dataset. For words whose length is less than max_word_length, zero padding is added. Note that while calculating the loss, you are supposed to exclude the padded instances from the loss computation.

The entire dataset is divided into minibatches with same dimension (add zeros for nonexistent letter in a word). This trick makes data loading/exporting easier, such as in `data_loader` and `get_conv_feature`. However, it is a little bit troublesome for crf because we have to clamp the zero-padded words. This results in one line of code for computing the valid lengths of words before feeding the whole batch into convolution layers.

Conditional Random Fields. The CRF model is the same as what was defined in Assignment 2. However, there is a difference in the input that is passed to the CRF model. To recall the details of the OCR dataset - the training set consists of n words. The image of the t-th word can be represented as $X^t = (\mathbf{x}_1^t, \ldots, \mathbf{x}_m^t)'$, where $'$ means transpose, t is a superscript (not exponent), and each *row* of X^t (*e.g.* $\mathbf{x}_m^t$) represents a letter. Here m is the number of letters in the word, and $\mathbf{x}_j^t$ is a 128 dimensional vector that represents its j-th letter image. To ease notation, we simply assume all words have m letters. The sequence label of a word is encoded as $\mathbf{y}^t = (y_1^t, \ldots, y_m^t)$, where $y_k^t \in \mathcal{Y} := \{1, 2, \ldots, 26\}$ represents the label of the k-th letter. So in Figure 1, $y_1^t = 2$, $y_2^t = 18$, …,

$y^t_5 = 5$.

In this assignment, the CRF model will instead take *convolutional features*, given by a function g, which you will implement. The details of the convolution operation is given in Section 3.

Using this (new) notation, the Conditional Random Field (CRF) model for this task is a sequence shown in Figure 2, and the probabilistic model for a word/label pair $(X, \mathbf{y})$ can be written as

$$p(\mathbf{y}|X) = \frac{1}{Z_X} \exp\left(\sum_{s=1}^{m} \langle \mathbf{w}_{y_s}, g(\mathbf{x}_s) \rangle + \sum_{s=1}^{m-1} T_{y_s, y_{s+1}} \right) \tag{2}$$

$$\text{where} \quad Z_X = \sum_{\hat{\mathbf{y}} \in \mathcal{Y}^m} \exp\left(\sum_{s=1}^{m} \langle \mathbf{w}_{\hat{y}_s}, g(\mathbf{x}_s) \rangle + \sum_{s=1}^{m-1} T_{\hat{y}_s, \hat{y}_{s+1}} \right). \tag{3}$$

$\langle \cdot, \cdot \rangle$ denotes inner product between vectors. Two groups of parameters are used here:

- **Node weight:** Letter-wise discriminant weight vector $\mathbf{w}_k \in \mathbb{R}^{128}$ for each possible letter label $k \in \mathcal{Y}$;
- **Edge weight:** Transition weight matrix T which is sized 26-by-26. T_{ij} is the weight associated with the letter pair of the i-th and j-th letter in the alphabet. For example $T_{1,9}$ is the weight for pair ('a', 'i'), and $T_{24,2}$ is for the pair ('x', 'b'). In general T is not symmetric, *i.e.* $T_{ij} \neq T_{ji}$, or written as $T' \neq T$ where T' is the transpose of T.

Given these parameters (*e.g.* by learning from data), the model (2) can be used to predict the sequence label (*i.e.* word) for a new word image $X^* := (\mathbf{x}^*_1, \ldots, \mathbf{x}^*_m)'$ via the so-called maximum a-posteriori (MAP) inference:

$$\mathbf{y}^* = \operatorname*{argmax}_{\mathbf{y} \in \mathcal{Y}^m} p(\mathbf{y}|X^*) = \operatorname*{argmax}_{\mathbf{y} \in \mathcal{Y}^m} \left\{ \sum_{j=1}^{m} \left\langle \mathbf{w}_{y_j}, g(\mathbf{x}^*)_j \right\rangle + \sum_{j=1}^{m-1} T_{y_j, y_{j+1}} \right\}. \tag{4}$$

When CRF is used as a layer, we need to compute the gradient with respect to its input. In (2), let us denote $z_s = g(\mathbf{x}_s)$. Then

$$\nabla_{z_s} p(\mathbf{y}|X) = \mathbf{w}_{y_s} - \sum_{\hat{\mathbf{y}} \in \mathcal{Y}^m} \frac{\exp(\ldots)}{Z} \cdot \nabla_{z_s} \sum_{s=1}^{m} \langle \mathbf{w}_{\hat{y}_s}, z_s \rangle \tag{5}$$

$$= \mathbf{w}_{y_s} - \sum_{\hat{\mathbf{y}} \in \mathcal{Y}^m} p(\hat{\mathbf{y}}|X) \mathbf{w}_{\hat{y}_s} \tag{6}$$

$$= \mathbf{w}_{y_s} - \sum_{\hat{y} \in \mathcal{Y}} p(y_s = \hat{y}|X) \mathbf{w}_{\hat{y}}. \tag{7}$$

2 PyTorch

You will use the popular PyTorch deep learning framework to implement all the algorithms in this assignment. For a comprehensive introduction to PyTorch please refer to this link PyTorch Tutorial.

The above tutorial is a beginner-friendly introduction to the basic concepts of PyTorch. You will need to be comfortable with all the concepts in that link to successfully complete this assignment. In particular, pay attention to the `nn.module` class. This is a standard way computational units are programmed in PyTorch. You will implement the `CRF` layer and the `Conv` layer, in the code, as a subclass of the `nn.module` class. Refer to the starter code for more details.

Like in the last assignment, Torch has a gradient checker. You could use it like this:

https://discuss.pytorch.org/t/how-to-check-the-gradients-of-custom-implemented-loss-function/8546

3 (20 points) Convolution

Different from the previous assignment, we are going to feed in convolutional features of the input image of a letter to the CRF model. Your task is to **implement the convolution layer in PyTorch**. Note that PyTorch implements its own convolution layer (`nn.conv2d`). You are required to provide your own implementation and **NOT** use PyTorch's implementation. However, you may use PyTorch's implementation of convolution as a reference to check the correctness of your implementation.

Convolution operation. Convolution is a commonly used image processing technique, applying various types of transformations on an image. Convolutional Neural Networks (CNNs) employ multiple layers of convolutions to capture fine-grained image features, which are further used downstream in learning several computer vision tasks such as object detection, segmentation etc.

A convolution operation takes in an image matrix X and a filter matrix K and computes the following function as detailed in Eq 9.6 of [GBC]:

$$\hat{X}(i,j) = \sum_{k,l} X(i+k, j+l)K(k,l). \tag{8}$$

In our case, the output channel from CNN is 1. Kernel is square: 5×5 or 3×3.

(3a) **(20 points)** Implement the `Conv` layer and the `get_conv_features(x)` function, in the starter code. Once convolution is implemented, the CRF's `forward` pass and `loss` functions use the convolution features as inputs (the code for this is set up already).

Your implementation also needs to accommodate different strides, along with an option of zero padding or not.

Testing your implementation. Your implementation of the `Conv` layer will contain the implementation of the convolution operation. It is crucial to get the implementation of the convolution operation correct first. Consider this simple example of an input matrix X and filter matrix K, with unit stride and zero padding. Report the result of convolving the X

with K. You must write this as a test case for the grader to run.

$$X = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix}; \quad K = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

Implement your test case inside a file `conv_test.py`. It should run as a standalone test (with all the dependencies, imports in place), and print the result on the screen.

Note. In PyTorch input to conv2d are 4-D tensors - (`batch_size` × `channel_size` × `height` × `weight`) for both the input image X and the filter K. In our dataset, we use a single channel input (`channel_size = 1`).

You only need to implement the forward pass of the convolution layer. There is no need to implement the derivatives. You can use PyTorch's auto-differentiation feature to automatically get backpropagation.

Having the `backward`() function implemented is an indication to PyTorch that the backward pass is indeed implemented. If a layer in the model is specified to have the `backward` function, then PyTorch will just use it. Otherwise, if some layers do not have the `backward` function explicitly implemented, then PyTorch will use autograd to compute the gradients.

[Answer] Convolution implementation Result:

tensor([[[[2., 2., 3., 1., 1.],

[1., 4., 3., 4., 1.],

[1., 2., 4., 3., 3.],

[1., 2., 3., 4., 1.],

[0., 2., 2., 1., 1.]]]], device='cuda:0', grad_fn=CopySlices)

PyTorch implementation Result:

tensor([[[[2., 2., 3., 1., 1.],

[1., 4., 3., 4., 1.],

[1., 2., 4., 3., 3.],

[1., 2., 3., 4., 1.],

[0., 2., 2., 1., 1.]]]])

You can run **conv_test.py** to see the result.

4 (50 points) CRF

Now, you will (re)-implement the CRF model in PyTorch. Note that this version is designed to use convolutional features and NOT the raw pixels to the CRF model (recall in Assignment 2 we used raw images pixels as the input features $\mathbf{x}_j^t$ which is a 128 dimensional vector). Here, they will be replaced by convolutional features. However, the CRF implementation should remain almost the same; except for changes in the input and output shapes and the fact that it needs to be implemented as a layer (`nn.module`) in PyTorch.

The CRF model is implemented as a `class` in the `crf.py` file in the provided starter code.

(4a) Implement the forward, backward pass and loss inside `crf.py`. This would amount to

1. re-implementing the *inference* procedure using dynamic programming (decoder)
2. dynamic programming algorithm for *gradient computation* including with respect to CRF input, T, and w_y,
3. loss - which is the negative log-likelihood of the CRF objective.

You can directly copy from the reference solutions for the last assignment or use your own implementation. Once again, place holders for all these are provided in the starter code (in the `crf.py` file). This question will be graded through the subsequent questions.

[**Answer**] We have implemented the **forward, backward** pass and **loss** in **crf.py**.

(4b) **(20 points)** Implement and display performance measures on the CRF model - we will use the same performance measures as the previous assignment (1) **letter-wise prediction accuracy**, (2) **word-wise prediction accuracy**. Using a batch size of 64 plot the letter-wise and word-wise prediction accuracies on both training and test data over 100 iterations (x axis). (Place holder provided in the startup code). Use a 5×5 filter matrix for this experiment, and set stride and zero/no padding to optimize the test performance. Initialize the filters randomly. If it has not converged (function value changes little), you may increase the number of iterations.

Note. Your model should process the input data batch after batch, therefore the input dimension to your model is 256*14*128 (`batch_size` * `max_word_length` * `num_of_pixels`). The output dimension of the CNN layer (i.e., the input dimension of the CRF layer) should be 256*14*64, where 64 is the length of the features of one letter.

Note. In the backward function of CRF, should we calculate and return the gradient of loss with respected to the whole batch of input, which is of dimension 256*14*64? We are calculating the gradient of loss with respect to weights. But the loss is calculated using the features and not the image itself.

Note. You have to make sure that invalid letters do not add any value to your loss function. Moreover, while you are calculating the gradient, you have zero-out the gradients of invalid letters. You cannot just use the true labels to find valid letters because, in test time, the labels are not provided to the model. The solution was to use the input image and find valid images (any image with at least one non-zero pixel). You can use 'torch.any()' and 'torch.where()' for masking and filtering.

[**Answer**] Parameteric values used:

Batch size = 64

Number of iterations = 100

C = 1000 for CRF

LBFGS lr = 0.001

Padding = zero

[Answer] Plots:

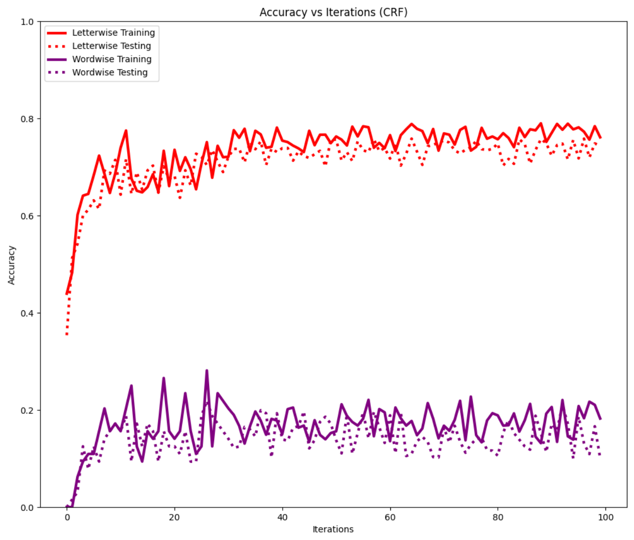


Figure 3. Letter-wise and word-wise accuracies using LBFGS optimizer over train and test data.

(4c) **(20 points)** It is common to use more than one convolution layer in modern deep learning models. Convolution layers typically capture local features in an image. Stacked convolutions (multiple layers of convolutions put one after the other) help capture higher level features in an image that has shown to aid classification significantly. Repeat experiments in (4b) with the following convolution layers. Set stride and zero/no padding to optimize the test performance.

1. A Layer with 5×5 filter matrix
2. A Layer with 3×3 filter matrix

Note. In `crf.py`, the `get_conv_features`() function is merely a "placeholder" for getting the convolution features for the CRF model. The output shape of the conv layer will vary depending on filter size, padding & stride. You are supposed to handle that (by zero padding) and make the input tensor to the CRF as a fixed shape, after you get convfeatures inside `get_conv_features`().

Note. In `train.py`, `conv_shapes` = [[1,64,128]] is a model parameter that specified the input/embedding shapes. Feel free to ignore this parameter and define your own way of handling the shapes. Here 64 is the embedding dimension, but if you are having multiple convolution layers (stacked convolution layers), then the shapes of the subsequent layers will be different. `Batch_size` is typically not specified in `conv_shapes` since its not a property of the layer; no matter what shapes you specify, the definition of `batch_size` will not impact them, the shape of any data (input/output) will be something like `batch_size` $\times$ a $\times$ b $\times$ c.

Note. In `train.py`, what does `embed_dim` = 64 mean? Typically layers are specified via the input and output dimensions that it produces. Here `input_dim` corresponds to the input dimension that the layer takes in and the `embed_dim` is the size of the embedding (output) that the layer produces. These are merely placeholders, meant to help you consider the input and output shapes while programming the CRF layer (and subsequently the conv layer). If this is confusing for you, feel free to setup your own mechanism to correctly handle input output shapes. The question asks you to perform convolution with different filter shapes. So one way to handle output shapes correctly is to let your convolution layer automatically infer the output shape, given the input shape and filter size; some pointers:

https://fomoro.com/projects/project/receptive-field-calculator

Note. Should we use Sequential to concatenate convolution layer and crf layer, or implement a CRF layer that entangles with a couple of convolution layers? Use Sequential. Pass a conv layer (it could be a stack of sequential conv layers) to CRF and use the `get_conv_features`() method to get the features. This way, we are able to easily determine the valid letters from the input image as well.

[**Answer**] Experimented same as (4b). Following are the parametric values used:

Stride = zero

Padding = zero

batch size = 64

Iteration = 100

C = 1000 for CRF

LBFGS learning rate = 0.001

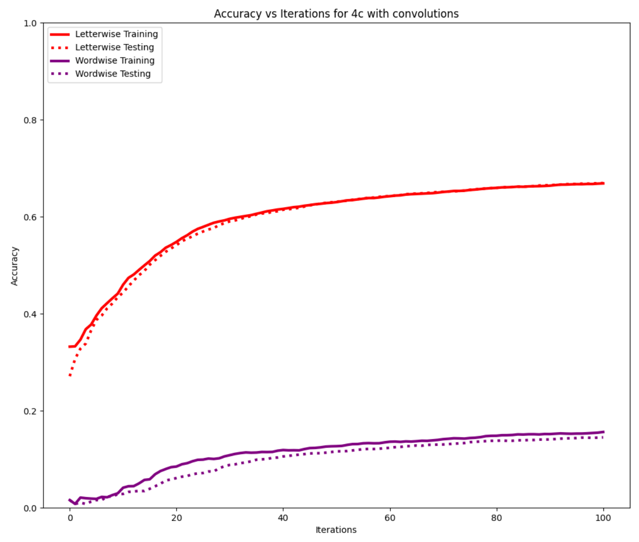


Figure 4. Letter-wise and word-wise accuracies using LBFGS optimizer over train and test data using CNN-CRF.

(4d) **(10 points)** Enable GPU in your implementation. Does it lead to significant speedup? You can test on the network in 4c. Make sure your plot uses wallclock time as the horizontal axis.

[Answer] Plot:

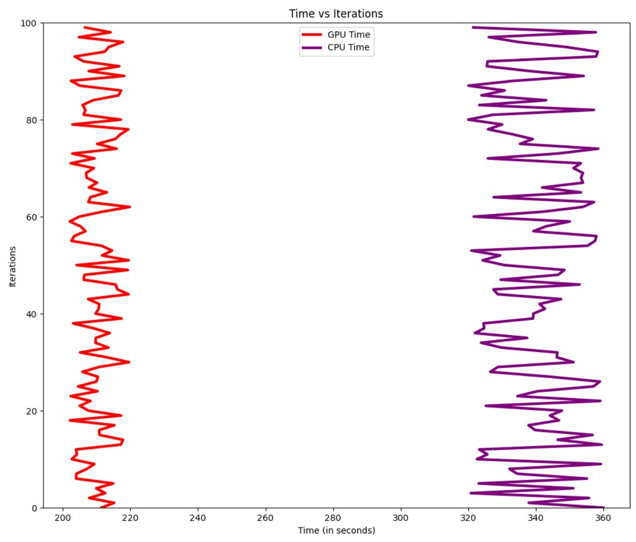


Figure 5. CPU vs GPU performance comparison over time using CNN (4c)

From the experiment we observed that GPU performance was somewhat slower than CPU performance. The reason GPU is a bit slower is due to heavy computation requirement in the function dp_infer() in crf_utils.py. Another reason CPU is faster due to smaller model size, and data transfer rate.

5 (30 points) Comparison with Deep Learning

Compare your new CRF model, with convolution features, with a convolution based Deep Neural Network (DNN) model of your choice, also known as Convolutional Neural Networks (CNN). You are free to design your own DNN model or pick one of the popular model architectures that have been published. Some popular choices would be,

1. VGG [4]
2. ResNet [1]
3. AlexNet [2]
4. GoogLeNet [5]
5. LeNet [3]

Since all these methods, except LeNet, require resizing the images from 16x8 to 224x224, you can just consider LeNet. You can use code from online, and build a blank LeNet and train all weights from scratch. It shouldn't be hard, and it will not take long. The input to the CNN model will of course be the original train and test dataset. None of these methods are composed with a CRF. You will have to report the following in your report.

(5a) **(10 points)** If you designed your own DNN model, then report the implementation details of it, along with the model architecture, loss functions etc. If you picked LeNet, explain each of the layers inside it and its purpose for the task at hand. That is, what functions (layers) were useful in the solution to the problem. In addition, look into the source code and sketch its structure

[Answer] We used LeNet-5 architecture to compare the new CRF model, having a total of 5 layers. There are 2 Convonutional(Conv2D) layers, 2 Fully-connected layers, and 1 Softmax layer with 26 node output representing 26 labels. Since we've a small dimensional image 16x8 therefore we don't need a deeper network, a few layers should be enough. The Conv2D layer creates a convolution kernel of 3x3 that is winded with layers input size of 16x8 image which helps producing an output tensor to feed the following layers. The Conv2D layer is followed by the Max-pooling layer to calculates the largest value in each patch of feature map which captures the important pixel values of an image. Then we have Fully-connected layer to multiply the input by a weight matrix in addition to a bias vector required for training. At the end we use softmax activation function in the output layer as we want to predict a multinomial probability distribution of 26 labels. Below is the architecture of the model:

(5b) **(5 points)** Plot the letter-wise and word-wise prediction accuracies on both training and test data (y axis) over 100 iterations (x axis) (You might have to implement these). Compare this with your CNN+CRF results and report your analysis (which model fared better? and why?). You may use the hyperparameter that yields the best performance for your CNN+CRF model. If it has not converged in 100 iterations, you may increase the number of iterations.

Sometimes, your program might run out of memory. In this case, you will have to adjust the batch size. This post might help:

https://stats.stackexchange.com/questions/284712/how-does-the-l-bfgs-work

[Answer] Parametric values for LBFGS optimization using LeNet:

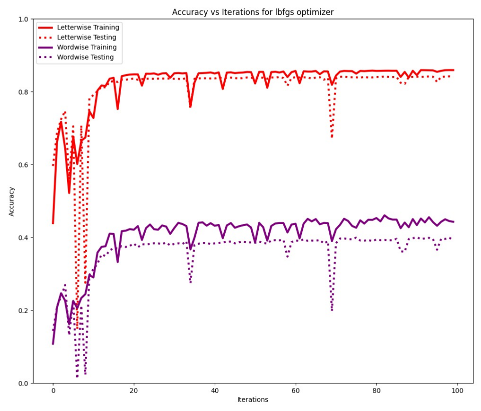
batch_size = 1200

train-test split 50%

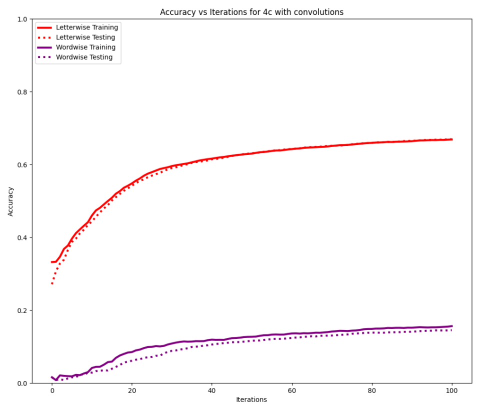
Gradient clipping : function clip_grad_value, clip_value=3.0

LBFGS with learning rate = 0.01

Plots:



(a) LBFGS



(b) CNN-CRF

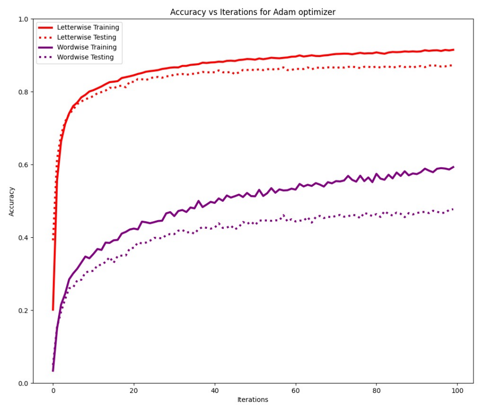
Figure 6. LBFGS vs CNN-CRF letter-wise and word-wise accuracies over test and train set

Using the above hyper-parameters it was observed that LBFGS outperformed the CNN-CRF model for both train and test data as it was able to converge faster and better compared to CNN-CRF. Tweaking hyper-parameters, and some regularizations didn't smoothen the results enough. For letter-wise prediction, LBFGS performed with an accuracy of 84% whereas, CRF was only able to achieve an accuracy of 67%.

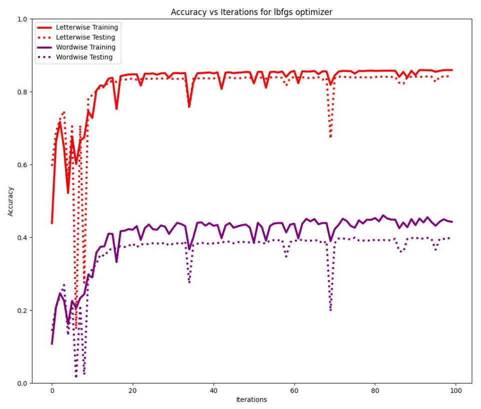
For LBFGS, we experimented with tweaking batch size to a large value and observed that a lot more data was required to train the model compared to the CRF model. It was observed that a batch size of 1200 was optimal. Another issue we faced during the training was getting NaN values while using LBFGS optimizer. To overcome this problem, we used Gradient-clipping. Results with different parameters can be seen in **LBFGS_FILES** folder.

(5c) **(5 points)** Change the optimizer from *LBFGS* to *ADAM*. Repeat the experiments in (5b) and report the letter-wise and word-wise accuracies, with x-axis as the #iterations. Does *ADAM* find a better solution eventually, and does it converge faster than *LBFGS*?

[Answer] Plots:



(a) Adam optimizer



(b) LBFGS optimizer

Figure 7. Adam vs LBFGS letter-wise and word-wise accuracies over test and train set

Comparing both the plots for letter-wise and word-wise accuracies using Adam optimizer and LBFGS optimizer, we observed that Adam tends to converge faster and finds a better solution faster compared to LBFGS. The reason LBFGS is a bit slower is due to its heavy second-order computation when the input size tends to increase. On the other hand Adam is a first order method which converges faster.

(5d) **(10 points)** Why did you choose this model (again it could be your own design or an off-the-shelf model)? More precisely you should explain every design decision (use of batchnorm layer, dropout layer etc) and how it helped in the task at hand, in your report.

[**Answer**] Since we've a small dimensional image 16x8 therefore we don't need a deeper network, a few layers should be enough. The Conv2D layer creates a convolution kernel of 3x3 that is winded with layers input size of 16x8 image which helps producing an output tensor to feed the following layers. We kept the number of channels as 3 and applied padding according to it. We perform batch normalization to avoid covariate shift and speed up the training process. The Conv2D layer is followed by the Max-pooling layer to calculates the largest value in each patch of feature map which captures the important pixel values of an image. Then we have Fully-connected layer to multiply the input by a weight matrix in addition to a bias vector required for training. At the end we use softmax activation function in the output layer as we want to predict a multinomial probability distribution of 26 labels.

The main reason for choosing LeNet is that LeNet perform well with small sized inputs (16x8 in our case).

Bibliography

[1] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition.

[2] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc.

[3] Yann Lecun, Leon Bottou, Y. Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86:2278–2324, 1998.

[4] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition.

[5] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going Deeper with Convolutions.