Contents

List of Figures

List of Tables

Chapter 1

Introduction

Deep Learning has enormously improved state-of-the-art techniques in many application areas today including object detection, image classification, speech recognition, sequence-modelling among others. However, not only this, deep learning techniques and architectures have achieved significant results in developing end-to-end systems for complex tasks such as generating realisitc images[], as well as generating pseudo-random numbers and learning to protect communication systems using neural networks. Besides, the different types of neural networks are nowadays being used for specific tasks such as recurrent neural networks being extremely successful in Natural Language Processing(NLP) tasks (such as machine translation, image/video captioning), Convolution Neural Networks(CNNs) being the most suitable for handling images and achieving better than state-of-the-art results in various image classification, recognition tasks including ImageNet.

1.1 Deep Learning and Cryptography

Chapter 2

Modified Convolutional Neural Networks

2.1 Introduction

Convolutional Neural Networks(CNNs) have been extremely successful at computer vision tasks since they were first used in 2012's ImageNet competition where CNNs significantly reduced the classification error from 26% to 15%. Since then, many CNN architectures have been proposed for various appolications and tasks. A major commonality among them is the ever-increasing number of layers, as the task becomes more and more complex along with an increasing availability of graphical processing units(GPUs) and devices with enough memory and computational power to effectively deploy deep architectures. In general, a Convolutional Neural Network consists of the following types of layers:

- **Convolution Layer**:

2.2 Label Consistency Module

As proposed in the paper,[], Label Consistency Module, is a slight modification proposed to the standard CNN architecture. In deep Convolutional Neural Networks, we generally delay the use of supervised information(i.e. the target output) until the final layer and thus, hidden layers only use the backpropagated gradients from the layers ahead. The gradients when propagated backwards tend to approach very small values for the starting layers thus leading to the vanishing gradient problem which tends to reduce the effectiveness of the training procedure. Besides, the use of supervised information is very minimalistic in such deep networks since only the output layer gets information about the actual class information and other hidden layers are trained without any guidance on the class information. Label Consistency Module proposes the use of explicit supervision with late hidden layers where a particular class label is associated with each neuron in the specific hidden layer so that the neuron is trained to activate only for data associated with that particular class label. A Label Consistency Constraint called the "Discriminative Representation Error" loss is incorporated into the overall objective function. An example of a Label Consistent Neural Network structure is shown in Figure 2.1

2.2.1 Training and Formulation

The overall objective function for Label Consistent Neural Network(LCNN, i.e. a neural network with the Label Consistency Module), is given as:

$$L = L_c + \alpha L_r$$

where, L_c is the classification error at output layer, α is a hyperparameter and L_r is the Discriminative Representation Error.L_r is calculated as:

$$N_l = \text{No. of hidden units in layer } l.$$

$$a^l = \text{Representation/Activations of Layer } l.$$

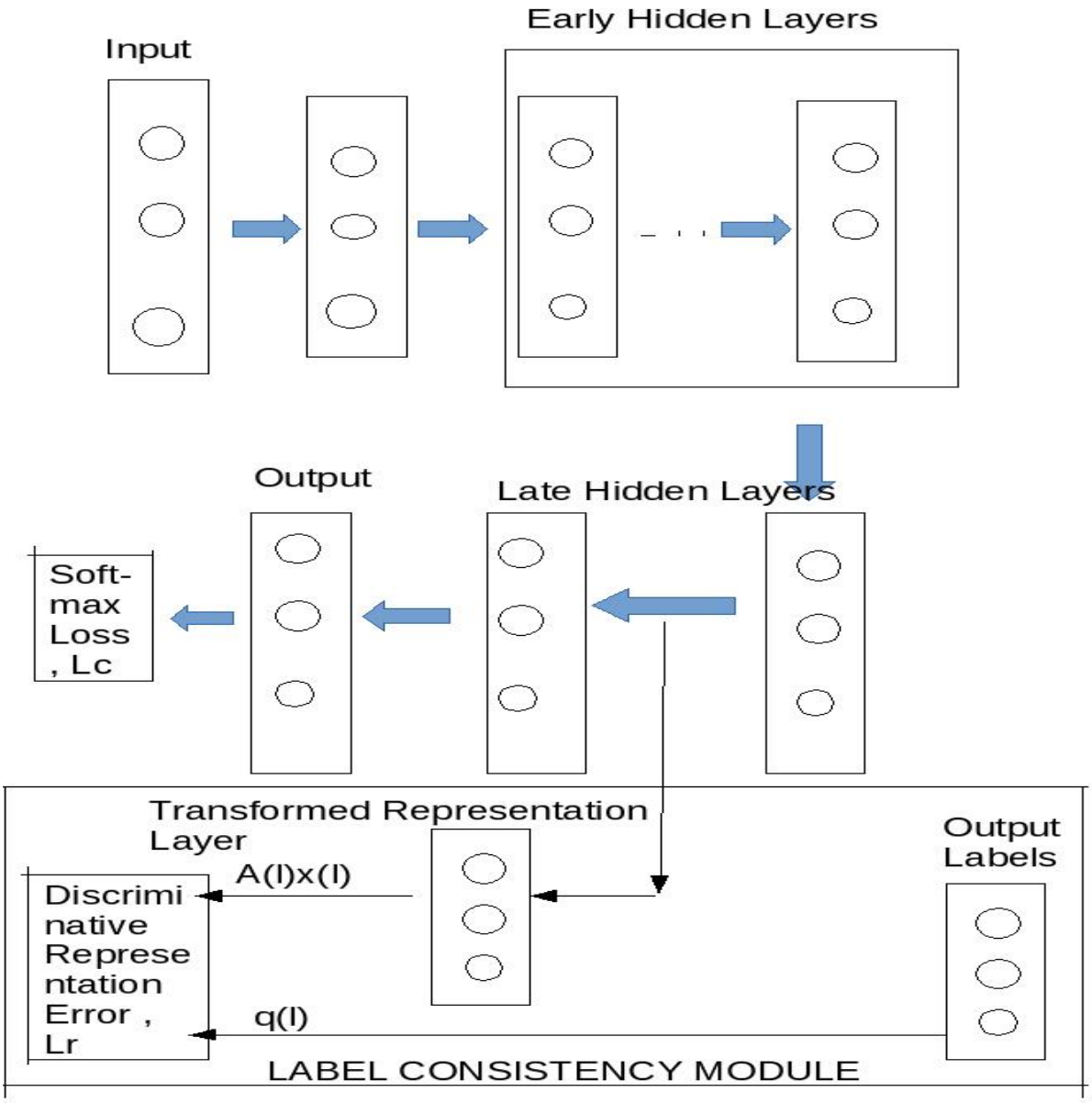

Figure 2.1 An example of LCNN architecture.[]

$$A^l = \text{Linear transformation Matrix, where } A^l \in R^{N_l \times N_l}$$
$$q^l = [q_1^l, q_2^l, \ldots, q_j^l, \ldots, q_{N_l}^l]^T \in \{0,1\}^{N_l}$$
$$L_r = ||q^l - A^l a^l||_2^2$$

q^l represents the ideal discriminative representation i.e. it represents the actual classes that each neuron in layer l should represent. Let m denote the number of training examples, $X = \{X_1, X_2, \ldots, X_m\}$ represent the training input and $Y = \{Y_1, Y_2, Y_3, \ldots, Y_m\}$ represent the corresponding output classes, then q^l is defined along with the training data as follows:

For each training example $i = 1$ to m:

$c = Y_i$

For $j = 1$ to N_l:

$q_j^l = 1$, if the j^{th} neuron has been assigned to class c.

$q_j^l = 0$, otherwise

In this way we have a matrix $Q^l = [Q_1^l, Q_2^l, \ldots, Q_m^l]$, where $Q_i^l = q^l$ for the i^{th} training example. Also, training samples which are from the same class would thus have similar representations.

Training is done using stochastic gradient descent as follows:

$$\frac{\delta L}{\delta a^i} = \begin{cases} \frac{\delta(L_c)}{\delta(a^i)}, & \text{if } i \neq l \\ \frac{\delta(L_c)}{\delta(a^i)} + \alpha \frac{\delta(L_r)}{\delta(a^i)}, & \text{if } i = l \end{cases}$$

where l is the layer at which the label consistency module is applied.

Now,we know that:

$L_r = ||q^l - A^l a^l||_2^2$

Hence,

$\frac{\delta(L_r)}{\delta(a^i)} = 2(A^l a^l - q^l)^T A^l$

Also, considering W to represent the weight matrices for the network:

$$\frac{\delta L}{\delta W^i} = \frac{\delta L}{\delta a^i}\frac{\delta a^i}{\delta W^i} = \frac{\delta L_c}{\delta W_i} + \alpha\frac{\delta L_r}{\delta W_i}$$

$$\frac{\delta L}{\delta W^i} = \frac{\delta L_c}{\delta W^i}, \qquad \forall i = \{1, 2, 3, \ldots, n\}, \text{ where } n \text{ is the no. of layers in the network}$$

$$\frac{\delta L}{\delta A^l} = \frac{\delta L_c}{\delta A^l} + \alpha\frac{\delta L_r}{\delta A^l} = \frac{\delta L_c}{\delta A^l} + 2\alpha(A^l a^l - q^l)(a^l)^T$$

$$\frac{\delta L}{\delta A^l} = 2\alpha(A^l a^l - q^l)(a^l)^T$$

2.2.2 Experimentation and Results

The Label Consistency Module greatly resembles the concept of sparse coding. The code for Label Consistency Neural Network used for experimentation is written in python using libraries tensorflow and keras and it is run on _____GPU processor.

2.3 Binary Denoising Convolutional Network

Denoising Convolutional Neural Networks is a relatively new technique that is basically used to prevent the problem of overfitting of data in deep CNNs. The idea is to basically corrupt the input data and train the network for the clean images as output data. Thus, the neural network would preferably learn a better feature representation for the input data. Besides, it would also deal with structured and unstructured noise present in the original image, thus acting as a method for removing noise from images as compared to the traditional methods which involved data augmentation as the primary basis for removing noise. Here, a binarized version of Denoising CNNs has been proposed in the paper,[] which first binarizes the image patches using some suitable threshold, then adds noise to it and finally trains on the noisy data with the target being the original clean image.

2.3.1 Training and Formulation

The network architecture for the denoising network proposed in the paper is shown in Figure 2.2 Using the architecture described above, first the input images are broken

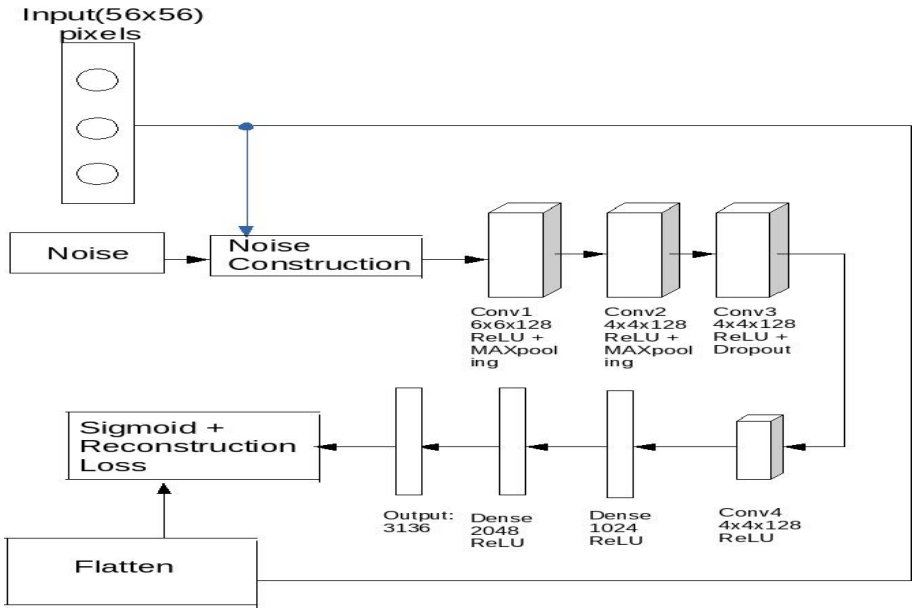

Figure 2.2 Binary Denoising Convolutional Networks.[]

down into patches of size 56X56 each with a overlap between each window of 10 (the paper also proposes patches of size 120x120 with overlap of 20), then these image patches are binarized using some threshold value(in the paper the proposed value is 0.45 of the document maximum but since in the IAM Handwriting dataset used for training the network images consist of large portions of white patches so a threshold value of 0.45 of document maximum would result in almost 99% values being 1 in most of the patches so instead a threshold value of 0.95 of the document maximum is used here so that only completely certain white patches are classified as 1 and even a slight amount of noise is classified with value 0). After binarization of the image patches, noise is added to these original input patches. To add noise, random images from the web as well as images from the own dataset are chosen as noise n_i, they are reshaped to the input size i.e.56x56 here and these random images are further augmented by changing the intensity and contrast. These noisy images are then used to add noise to the original image patches as follows:

$$g(\text{ input\_patch},n_i) = \frac{1}{255} min(2 * 255 - \text{ input\_patch - } n_i * 255)$$

Finally the noisy image patches are then trained using the denoising architecture shown in Fig.2.2 with the target output being the clean binarized image patches. The loss function is modified a bit to incorporate the possibility of completely white patches or patches with only structured noise as follows:

$$\mathcal{L} = -\sum_{i}^{batch\_size}(1 + \lambda\mathcal{I}(x_i))(x_i)^T log(f(g(inputpatch, n_i)))$$

where, λ is a hyperparameter, x_i represents the flattened target output, *inputpatch* represents the 56x56 input image patch, n_i represents the noise added to the i^{th} input image patch, *batch_size* represents the batch size for batch gradient descent, $f(.)$ represents the overall binary denoising convolutional network architecture, $g(.)$ represents the noise function as described above and $\mathcal{I}$ represents an indicator function which denotes when the input patch is a patch with no handwriting i.e. a completely white patch or a patch with only structured noise. This function is implemented here by determining the percentage of 1's in the binarized image patch.

As suggested in the paper, Nesterov Momentum update is used for the gradient descent training algorithm. After training the above network, the output of the binary denoising convolutional network is binarized using a proper threshold value. Now, the two-pass algorithm is implemented to determine the connected components in this binarized image. The obtained connected components are now passed through thresholds on both aspect ratio and the size(i.e. no. of pixels). The paper further suggests data augmentation steps and then using an architecture similar to Fiel et.al.[], for writer identification on image patches which are then merged using a global classifier but these haven't been implemented here.

2.3.2 Experimentation and Results

Chapter 3

Neural Networks for Resource-Constrained Embedded devices

3.1 Introduction

Embedded systems are basically computer systems with special-purpose computing goals, i.e. they perform a dedicated function within some larger system and thus, they are often embedded into some complete device. These systems have real-time computing constraints and have limited memory. Such devices include cell phones, printers, calculators, handheld computers etc. With the growing popularity of deep neural networks, the computational resources required to run such powerful deep architectures have also increased, however, the further advent of Internet of Things(IoT) has , on the other hand, increased the need to implement these deep learning algorithms on the resource-constrained embedded devices. This requires reducing the space and time requirements of the deep learning architectures while maintaining the efficiency and accuracy of them. A typical Convolutional Neural Network(CNN) has around millions of parameters along with an even more number of arithmetic oper-

ations being performed and the trend is always towards trying to increase the no. of layers to learn more complex discriminative features. In order to deploy such a network on smart mobile devices, it is necessary to reduce the no. of bits required to store the parameters as well as reduce the no. of arithmetic operations required for training and testing the network, which can then be used for various tasks such as speech-driven Personal Assistant devices, always-on computer vision applications, face detection using high-resolution cameras among others.

In neural networks, multipliers are generally the most space and time-hungry components so an approach to reduce the space and power requirements of neural networks would be to reduce the no. of multiplications required for computations in the neural network and possible replace them with additions or XOR operations. Further the recent approaches towards reducing the computational requirements of neural networks include model pruning, which is a bit similar to the concept of dropouts, where edges with magnitude of weight smaller than a particular value are removed from the model; weight compression where a specific no. of bits are used for representing the weights which reduces the memory requirements of the neural network, some techniques also involve compression of both weights and activations. Other methods adopted for reducing the size of Deep Neural Networks include tensorizing and parameter quantization. Many authors have used weight compression techniques to reduce the size of Deep Neural Networks including binarization of weights as well as ternary weights. Courbariaux et.al.[] proposed the BinaryConnect algorithm where only binary values for weights were used i.e.-1 and 1 and a sign() function was used to binarize the weights. Further, binarization of activations was first proposed by Hubara et.al.[]. These weren't approximations of standard neural networks rather were separate networks allowing only binary weights, thus they weren't as successful on large datasets. Further, Rastegari et.al. proposed Binary Weight Networks and XNOR-Net as an approximation to standard CNNs by using scaling factors along with

binarized weights to approximated the convolution operation with simple XNOR operations, thus removing the need for multiplications. In XNOR-Net input signals were also binarized. Further ternary-weight-networks were also proposed with the weights taking values -1,0,1[], thus increasing the precision of bits as well as the memory requirements as compared to binary weights. Here, the ideas and implementations of 3 proposed techniques are discussed- Two-Bit Networks, Loss Aware Binarization and Bitwise Neural Networks.

3.2 Two-Bit Networks

Two-Bit Networks for resource-constrained embedded devices have been proposed in the paper,"Two-Bit Networks for Deep Learning on Resource-constrained Embedded Devices"[]. The idea of two-bit networks is basically an extension of the idea of ternary-weight networks which use 2 bits to represent values of weights in the neural network. Two-bit networks constrain weights to 4 values (-2,-1,1,2) using the same no. of bits as ternary-weight networks. Mutiplication operations are now reduced to addition, subtraction and shift operations.

3.2.1 Training and Formulation

Two-Bit Networks approximate the convolution operations with simple addition and subtraction operations, so the amount of power required for forward propagation in the network is drastically reduced. So:

$$L = \text{No. of layers in the network}$$

$$K_l = \text{No. of filters in the } l^{th} \text{ layer}$$

$$W^l = \text{Filters for the } l^{th} \text{ layer, } \forall l = [1, 2, \ldots, L]$$

$$\hat{W}^l = \text{Binarized filter for the } l^{th} \text{ layer, } \forall l = [1, 2, \ldots, L]$$

$$\alpha^{lk} = \text{Scaling Factor for the } k^{th} \text{ filter in } l^{th} \text{ layer}$$

$$w^l = \text{width of filters for layer } l$$
$$h^l = \text{height of filters for layer } l$$
$$w_{in}, h_{in} = \text{width and height for input layer respectively}$$
$$n_{lk} = \text{No. of elements in } k^{th} \text{ filter at layer } l$$

Hence, we have $W^l \in \mathcal{R}^{w^l \times h^l \times K^{l-1} \times K^l}$ $\forall l = [1, 2, \ldots, L]$.

So for the first convolution layer, let $I \in \mathcal{R}^{w_{in} \times h_{in} \times K^0}$ represent the input tensor, then the convolution operation is given by:

$$I * W^{(1)} \approx concatenate\_over\_k(I * (\alpha^{1k}\hat{W}^{(1)}[:,:,:,k])) =$$
$$concatenate\_over\_k((\alpha^{1k}I) * \hat{W}^{(1)}[:,:,:,k])$$

Now since the weights $\hat{W}^l$ are binary weights with only 4 possible values i.e. (-2,-1,1,2), so we can replace the multiplications for each filter in the layer with simple addition, subtraction and shift operations as (say multiplication of the $[x1, y1, z1]^{th}$ element in the k^{th} filter with the $[x, y, z]^{th}$ element in the new input tensor $\hat{I} = \alpha^{1k}I$):

$$\hat{W}^{(1)}[x1, y1, z1, k] \times \hat{I}[x, y, z] = \begin{cases} -(\hat{I}[x, y, z] << 1 \text{or } (\hat{I}[x, y, z] + \hat{I}[x, y, z])) & \text{if } \hat{W}^{(1)}[x1, y1, z1, k] = \\ -\hat{I}[x, y, z] & \text{if } \hat{W}^{(1)}[x1, y1, z1, k] = \\ \hat{I}[x, y, z] & \text{if } \hat{W}^{(1)}[x1, y1, z1, k] = \\ (\hat{I}[x, y, z] << 1 \text{or } (\hat{I}[x, y, z] + \hat{I}[x, y, z])) & \text{if } \hat{W}^{(1)}[x1, y1, z1, k] = \end{cases}$$

The same way we can convert all the multiplication operations during forward propagation in addition, subtraction and shift operations for all the filters for all layers i.e. $\forall k = [1, 2, \ldots, K_l]$, $\forall l = [1, 2, \ldots, L]$.

Now, the approximation of weights can be done provided we minimize the absolute difference between W_k^l and $\alpha^{lk}\hat{W}_k^l$ for $k = [1, 2, \ldots, K_l]$, i.e. we minimize the L2-norm of the error for each filter in each layer as:

$$(\alpha^*)^{lk}(\hat{W}^*)^l_k = argmin_{\alpha^{lk},\hat{W}^l_k}(||W^l_k - \alpha^{lk}\hat{W}^l_k||^2_2)$$
$$\text{where, } \alpha^{lk} > 0 \text{ and L2-norm is over } i = [1, 2, \ldots, n_{lk}]$$

Now, after substituting the binarized weights in terms of the actual weights, we finally obtain the optimal scaling factor for the k^{th} filter in the l^{th} layer as:(For full proof refer[])

$$(\alpha^*)^{lk} = \frac{\sum_{i\in B_1}|(W^l_k)_i| + 2\sum_{i\in B_2}|(W^l_k)_i|}{|B_1| + 4|B_2|},$$
$$\text{where } B_1 = \{1 \le i \le n_{lk} \mid |(W^l_k)_i| \le 1\},\ B_2 = \{1 \le i \le n_{lk} \mid |(W^l_k)_i| > 1\}$$

Stochastic Gradient Descent is used for training the two-bit network where forward and backward propagation is done using the binarized weights and the optimal scaling factors, however since the approximated binarized weights are very small, during the updating of weights using the SGD update rule, the real-valued weights are updated and not the binarized weights. So, first we initialize our real-valued weights randomly, then we compute our binarized weights using the discretization function:

$$\hat{(}W^l_k)_i = \begin{cases} -2 & \text{if } (W^l_k)_i < -1 \\ -1 & \text{if } -1 \le (W^l_k)_i \le 0 \\ 1 & \text{if } 0 < (W^l_k)_i \le 1 \\ 2 & \text{if } (W^l_k)_i > 1 \end{cases}$$

Then, the optimal scaling factors are computed for each filter of each layer using the formula given earlier. Now, forward propagation is done by replacing the multiplication operations as explained earlier with addition, subtraction and shift operations using the binarized weights and optimal scaling factors. Now back-propagation is performed using the binarized weights to obtain the corresponding gradients. Now, the corresponding real-valued weights are updated for every binarized weight using SGD update rule(momentum update rule is used in the paper).

3.2.2 Experimentation and Results

3.3 Loss-Aware Binarization of Deep Networks

This paper considers the effect of binarization on the loss function along with reduction in multiplications, by binarizing the weights which replace the underlying multiplications with additions, shifts and simple XNOR bit operations. Basically a proximal Newton algorithm is used with diagonal Hessian approximation for minimzing the loss w.r.t. the binarized weights. In general, deep networks are difficult to train due to the final objective function being highly nonconvex which makes it difficult to obtain a global minimum and increases the chances of getting stuck at a local minima. Hessian-free optimization is a second-order method which is based on the Newton's method for second-order minimization which basically approximates a function upto its second order derivative in the Taylor series expansion of the function, but eliminates the need for the Hessian matrix of second derivatives using conjugate gradient descent. Hessian-free optimization is one of the methods which has been used to alleviate the problem of training deep networks, along with element-wise adaptive learning as used by most of the recent update rules such as Adadelta, RMSProp and Adam(which also uses adaptive learning for equilibration).

3.3.1 Training and Formulation

Here, binary weights are considered, so:

$$W_l = \text{Weights for layer } l$$

$$L = \text{Total no. of layers}$$

$$n_l = \text{no. of nodes/neurons in layer } l$$

$$\hat{W}_l = \text{Approximated weights for layer } l$$

$$\alpha_l = \text{Scaling Factor for layer } l$$

Here, $\hat{W}_l = \alpha_l b_l, \alpha_l > 0, b_l \in \{\pm 1\}^{n_{l-1} \times n_l}, l = 1, 2, \ldots, L$, ($n_0$ represents the no. of

input neurons).

The proximal Newton Algorithm is generally used for solving composite optimization problems of the form :

$$min_x f(x) + g(x)$$

where f is convex and smooth whereas g can be noncovex or nonsmooth[]. The proximal Newton algorithm gives the solution as:

$$x_{t+1} = argmin_x(\bigtriangledown f(x_t)^T(x - x_t) + \tfrac{1}{2}(x - x_t)^T H(x - x_t) + g(x))$$

Using the proximal Newton Algorithm for minimizing our loss function we have:

$$l(\hat{(}W)^t) \approx l(\hat{(}W)^{t-1}) + \bigtriangledown l(\hat{W}^{t-1})^T(\hat{W}^t - \hat{W}^{t-1}) + \tfrac{1}{2}(\hat{W}^t - \hat{W}^{t-1})^T H^{t-1}(\hat{W}^t - \hat{W}^{t-1}))$$

Hessian matrix above helps in preconditioning i.e. making curvatures on the loss function similar along all directions. However, it is extremely time ans space-inefficient to compute Hessian matrices for deep networks so, the Hessian matrix is approximated by a diagonal positive definite matrix D. The recent stochastic optimization algorithms which pre-parameter adaptive learning method for preconditioning provide a good approximation for $diag(H^2)$, in the form of the second order gradient of moment, i.e. v in these algorithms. So, D is approximated using $\sqrt{v}$ for the gradients in each layer, so D_l is approximated using v_l, obtained as a decaying sum of second-order gradients of weights for layer l.

The paper provides a few results for the optimal scaling factor as:

$$W_l^t = W_l^{t-1} - \bigtriangledown_l l(\hat{W}^{t-1}) \oslash d_l^{t-1},$$

where d_l^{t-1} represents $diag(D_l^{t-1})$ i.e. the diagonal matrix for the Hessian approximation of layer l at time $t-1$ and $\oslash$ represents element-wise division.(The above update rule is based on Newton's method which is:

$x_{n+1} = x_n - (H(f)(x_n))^{-1} \bigtriangledown f(x_n)$ for $f : \mathcal{R}^n \to \mathcal{R}$, with D being a diagonal approximation of the Hessian matrix).

$$\alpha_l^t = \frac{||d_l^{t-1} \odot W_l^t||_1}{||d_l^{t-1}||_1}, b_l^t = sign(W_l^t)$$

Again during feed-forward propagation, multiplications operations can be converted to simple addition and subtraction operations. During feed forward propagation, first the optimal scaling factor is computed for each layer as well as the binarized weights b_l^t are calculated using the sign() function. Now the layer-l input is modified as $\hat{x}_{l-1}^t = \alpha_l^t x_{l-1}^t$. Now, the output of layer l can simply be calculated from $\hat{x}_{l-1}^t$ and b_l^t using simple addition and multiplication operations since $b_l^t \in \{\pm 1\}^{n_{l-1} \times n_l}$, so for $(b_l^t)_{ij} = -1$, we perform a subtract operation and for $(b_l^t)_{ij} = -1$, we perform an addition operation. Now we apply batch-normalization and the non-linear activation to the obtained output to get the final output for layer l. After feed-forward propagation, the gradients are computed w.r.t. the approximate weights i.e. W_l^t and then Adam update rule is applied where the diagonal matrix d_l^t is approximated using the second moment obtained using Adam update and the real-valued weights are updated using the equation specified earlier. The Loss Aware Binarization Scheme can also be extended to Recurrent Neural Networks just like above only that now, the weights between the recurrent units will also be binarized and trained just like other weights in the network.

3.3.2 Experimentation and Results

3.4 Bitwise Neural Networks

Bitwise Neural Networks were proposed in the paper "Bitwise Neural Networks"[]. These networks consider binarization of all the parameters of the network as well as binarization of the inputs, biases as well as activations of each layer. The idea is to replace all multiplications as well as addition operations with simple XNOR and bit counting operations. Besides all the parameters as well as inputs, biases and activations of intermediate hidden layers are represented using single bits. Besides, an

extended version of Bitwise Neural Networks is proposed in the paper where inactive weights are allowed i.e. some weights can be set to 0 besides, the binary weights. Besides, any boolean function which maps binary inputs with binary outputs only requires a single hidden layer(where each hidden neuron can be used to represent a possible mapping between the input and output patterns).

3.4.1 Training and Evaluation

First the parameters of the network are initialized using real-values between -1 and 1. The real-valued network is trained using weight compression to convert the real-valued network to a Bitwise Neural Network(BNN). The inputs are either single bits or real-valued between -1 and 1.Consider,

$$L = \text{No. of layers in the network.}$$
$$K_l = \text{No. of neurons in layer } l,\ K_0 \text{ represents the no. of input neurons}$$
$$W_{ij}^l = \text{Weight between the } i^{th} \text{ neuron in layer } l-1 \text{ and the } j^{th} \text{ neuron in layer } l$$
$$b_i^l = \text{Bias of } i^{th} \text{ neuron in layer } l$$
$$h_i^l = \text{Output of } i^{th} \text{ neuron in layer } l$$

Forward Propagation for training this real-valued network is just the standard forward propagation, only now we use $tanh()$ as the activation function so that the inputs to each layer are squashed between -1 and 1. So, we have

$$h_i^l = tanh(tanh(b_i^l) + \textstyle\sum_j^{K_{l-1}}(tanh(W_{ji}^l)h_j^{l-1})$$
$$\text{where } h^0 \text{ represents the input layer and } W^l \in \mathcal{R}^{K_{l-1}\times K_l}$$

Similarly, during backpropagation is similar to the standard procedure, only the derivative of the activation function changes:

$$\delta_j^l(Cost) = (\textstyle\sum_i^{K_{l+1}} tanh(W_{ji}^{l+1})\delta_i^{l+1}(Cost))(1-(h_j^l)^2)$$

Similar changes occur in the gradients of weights and biases w.r.t. the Cost.

After training the real-valued network, the actual Bitwise Neural Network is initialized

using the compressed weights learnt during the training of the real-valued network. This is done using a sparsity parameter,λ (which is the extended version for the network which involves inactive weights). Using λ, we determine the count of zero weights after binarization as $\lambda K_{l-1} K_l$ for W^l. Now, using binary search we find the value of a hyperparameter, say β, which controls the boundaries for binarizing the weights. So that (say the binary weights are represented as $\hat{W}^l$:

$$\hat{W}_{ij}^l = \begin{cases} -1 & \text{if } W_{ij}^l < -\beta \\ 0 & \text{if } -\beta \leq W_{ij}^l \leq \beta \\ 1 & \text{if } W_{ij}^l > \beta \end{cases}$$

The same is done for the biases to get binarized biases. Now, the obtained binary weights and binarized biases are used to initialize the weight and bias parameters of the bitwise network and then do noisy backpropagation to train this BNN. This is done as follows:

$$z_i^l = sign((b_i^l) + \sum_j^{K_{l-1}}((\hat{W}_{ji}^l) \otimes z_j^{l-1})$$

where z^0 represents the binary input layer and $\hat{W}^l \in \mathcal{B}^{K_{l-1} \times K_l}$

and z^l represents the binarized outputs of the l^{th} layer, $\mathcal{B} \equiv \{\pm 1, 0\}$ and $\otimes$ represents the XNOR operation.

Also the loss function is modified as:

$$\xi = \frac{\sum_i^{K_{L+1}}(1 - t_i \otimes z_i^{L+1})}{2},$$

where t_i represents the i^{th} value in actual output

K_{L+1} represents the no. of output neurons

Now, backpropagation is similar to the standard backpropagation where the activation function is now the $sign()$ function, so:

$$\delta_j^l(Cost) = \sum_i^{K_{l+1}} (\hat{W}_{ji}^{l+1} \delta_i^{l+1}(Cost))$$

Similarly the update rules for the weights and biases can be obtained, but again we don't update the binary weights and biases rather the real-valued weights and biases which we started off with are updated with the corresponding gradients of the binary weights and biases respectively and then binary values are recomputed using the sparsity parameter as earlier after every epoch.

3.4.2 Experimentation and Results

Chapter 4

Pseudo-Random Number Generators using Neural Networks

4.1 Introduction

Pseudo-Random Number Generators(PRNGs) form an integral part of cryptographic systems and play an important role in various cryptographic applications such as data encryption as well as approximating one-time pads along with stream ciphers, where in many crypto-mechanisms PRNGs are necessary to ensure randomness and unpredictability and thus provide security for the encryption transformations. In general, random number generation is required in various other physical as well as statistical simulations and thus, it is very important to have good PRNGs which should be secure, unpredictable as well as efficient. The present methods for random number generation are mostly based on deterministic algorithms due to the need to repeat the pseudo-random data in many applications of random number generators. Some of most prevelant methods for Pseudo-random number generators today are Blum-Blum Shub and Mersenne Twister Algorithms. The PRNGs take a random initial seed on which they build up a pseudo-random sequence. Due to the initial seed, the PRNGs donot remain truly random since being deterministic functions, the same seed would

always produce the same sequence.

The idea of PRNGs is to map the initial random seed to a longer pseudo-random string. Consider S to be the state space for PRNGs (extremely large in size), then PRNGs consist of a function $f : S \to S$ mapping the current state to the next state and a function $g : S \to \mathcal{U}$ where $\mathcal{U}$ is the output space which determines the output from the current state at each time. A probability distribution is also present which is used to decide the initial seed(state) from among set S, to ensure random initial state. For e.g.- $S = \mathcal{Z}$,$\mathcal{U} = \mathbb{R}$, i.e. the set of integers and real numbers respectively for Linear Congruential Generators which are a member of the larger class of Multiple Recursive Generators which form an important class of Random Number Generators. In cryptography, pseudo-random number generators are generally used with binary data hence, it specifies $f : \{0,1\}^l \to \{0,1\}^n$ i.e. the state space here consists of all binary strings of all possible sizes and $l \leq n$ so that a longer binary string is produced as output. Besides, we also have adversaries defined as $\mathcal{A} = \{A : \{0,1\}^n \to \{0,1\}^*\}$, and the PRNG is said to be a good PRNG, if it is able to fool the adversary in the sense that the output produced by the PRNG for a uniform distribution of length l is with ϵ-distance of the output produced by the adversary for the same uniform distribution of length n, thus the output produce by PRNGs should be computationally in-distinguishable from random sequences. Besides, in cryptography, we require unpredictability so ϵ should be almost negligible, and also the function f should be computable in polynomial-time but not invertible in polynomial-time.

A major area which is currently under research, is the use of neural networks to generate pseudo-random numbers. A major reason for consideration of neural networks for generating random numbers is the highly non-linear and complex nature of neural network layers which makes it possible to feed-forward propagate the input but is computationally infeasible to determine the initial input given the output random sequence produced by the neural network, thus resembling one-way functions. Also,the

time to crack the overall algorithm learnt by the neural network grows exponentially as the number of parameters in the network are increased, thus ensuring security of the learnt network. Besides the dynamics of neural networks have been used for random number generation with random orthogonal weight matrices and have proven successful[]. The theories of learning and memory in the brain also inspire using neural networks as random number generator since the weights between neurons in the brain are either strengthened or weakened depending upon the present situation as well as the frequency of the neurons firing together which overall produces different results in different situations.

Here a brief overview of a paper on using Binary Recurrent Neural Networks for random-number generation.

4.2 Pseudo Random Number Generation using Binary Recurrent Neural Networks

The paper[], proposes using binary recurrent neural networks for generating random numbers based on the concept of plasticity in neural networks which affects the strength of connections between neurons based on the way they interact. Basically two types of plasticities are considered:

- Intrinsic Plasticity : This basically tries to ensure that all the neurons in the network should fire at the same average activity rate. This is done by modifying the thresholds of individual neurons as compared to the other plasticities which consider modifying the synaptic connections between neurons based on the strength of their connection. So, if a neuron acquires a high pre-activation sum, automatically the threshold for that neuron would also increase to ensure an average firing rate for all neurons.

- Anti-Spike Timing Dependent Plasticity(Anti-STDP): This affects the synaptic weights between neurons and on the contrary, it weakens the connections between neurons which tend to fire at consecutive timesteps and instead, strengthen the connections between neurons firing at different timesteps. The paper specifies that this kind of update function for networks, trained using anti-STDP lead the network into a chaotic behaviour and tend to prevent the same outputs from repeating themselves frequently, i.e. the same set of neurons would tend to be activated only after a long cycle due to the chaotic behaviour, however this does require well-chosen parameters.

The concepts of Intrinsic Plasticity and Anti-STDP are used for updating the weights between neurons in the neural network.

4.2.1 Training and Formulation

The network is a fully-connected recurrent Neural network with neurons taking binary values i.e. 1 and 0 denoting whether they are activated(on) or deactivated(off). The feedback connection is between the hidden layer and the previous and previous-to-previous input layer and the output at timestep t becomes the input of the network at timestep $t+1$. Consider,

$$x(t) = \text{Input at timestep } t = \text{Output at timestep } t-1$$

$$k = \text{No. of neurons active at any time}$$

$$N = \text{No. of neurons in the input and hidden layers}$$

where $x(0)$ represents the initial random seed used as input, $x(0)$ is chosen as a random distribution on $(0, 1)$, k highest values in $x(0)$ are assigned value 1 and the rest $N - k$ are assigned value 0.

$$h(t+1) = \text{hidden layer activation at timestep } t+1$$

$W_{ij}(t)$ = Synaptic weight between j^{th} neuron in hidden layer and i^{th} neuron in input layer, $W_{ij} > 0$ and $W_{ii} = 0$

$T_i(t)$ = Threshold of i^{th} unit in hidden layer at timestep t

So, we have our update function as:

$$h_i(t+1) = (\sum_j^N W_{ji}(t)x_j(t)) - T_i(t) - max(x_i(t), x_i(t-1))$$

$x(t+1) = KWTA(h(t+1))$, where $KWTA$ represents the function which makes the k highest values in $h(t+1)$ as 1 and the rest as 0

Thus, if a neuron has been active in the previous two-timesteps then, it is prevented from firing in the current time-step. Using this two-step refractory period is inspired from biological neurons which go through periods of complete inactivation as well as periods of complete activation. Now, the update functions for the threshold values are given using Intrinsic Plasticity which tries to ensure an average firing rate of all neurons:

$T_i(t+1) = T_i(t) + \eta_{IP}(x_i(t) - k/N)$, where η_{IP} is a hyperparameter depicting the learning rate for Intrinsic Plasticity.

The update rule for the synaptic weights is given using anti-STDP with the aim of discouraging continous activations of neurons:

$\triangle W_{ji} = -\eta_{ASTDP}(x_i(t)x_j(t-1) - x_j(t)x_i(t-1))$, where η_{ASTDP} is a hyperparameter depicting the learning rate for Anti-Spike Timing Dependent Plasticity.

Now the next task is to generate pseudo-random numbers from the state/output vector of the network at any timestep. A particular way of doing so is to do bit sampling i.e. generate a single bit from the state vector of the system at a particular timestep and concatenate thse bits to produce a random sequence. Another method for generating pseudo-random numbers from the output of the network is the use of a parity gadget[]. The idea is to first get a parity-block from the network output in the

form of q neurons from the output. This is done using a shifting parity-block which is shifted ahead after some no. of steps. Now, for the parity block, the total no. of neurons are determined which were active during the previous timestep,:

$$x_{k_l} = \begin{cases} 0 & \text{if } \sum_{n \in \text{parityblock}} x_n(t-1) < l \\ 1 & \text{if } \sum_{n \in \text{parityblock}} x_n(t-1) \geq l \end{cases}$$

$q =$ No. of neurons in the parity block

Here, l denotes the threshold for no. of active neurons, $l = [1, 2, \ldots, q]$

$x_{k_l} =$ value of the output for $l = [1, 2, \ldots, q]$, considering l neurons to be active in the parity block furing theori previous time-step

Now, a parity neuron x_p is updated as:

$$x_p = \begin{cases} 0 & \text{if } (\sum_{l=1}^{q} x_{k_l}) mod 2 = 0 \\ 1 & \text{otherwise} \end{cases}$$

This parity bit is then appended to the random bit sequence and the process is repeated again, we get the required size of random sequence that we want, generally we require an 8-bit byte to form an integer in the range $[0, 255]$, so the no. of iterations is generally 8. However, we can generate a very long random bit sequence using the parity gadget. Besides, the determination of the parity bit leads to a one-way function thus, ensuring security of the pseudo-random numner generator thus produced. The last task is to test the randomness of the pseudo-random number produced. There are various methods which are used to test the randomness of PRNGs. NIST(National Institute of Standards and Technology) tests are the most preferred tests

used to check randomness however, they are very hardware specific and impracticality in implementation, first the PRNGs are tested using other methods such as DIEHARD, CRYPTX and ENT.

4.2.2 Experimentation and Results

Chapter 5

Neural Networks in Cryptography

5.1 Introduction

While deep neural networks have been used for various applications ranging from image processing tasks to speech related tasks as well as Natural Language Tasks. However, one of the major areas where recent advances have taken place, is Cryptographic Applications of Neural Networks. Using neural networks to learn encryption functions and decryption functions while ensuring the secrecy of the learnt network is one of the major applications of neural networks in cryptography which has gained attention lately. Neural Networks are fully capable of learning to use secret keys to protect their information from other neural networks. A general cryptosystem consists of two parties, say *Alice* and *Bob* communicating with each other in order to exchange information while a third party called the adversary, say *Eve* wishes to interrupt this communication and learn the messages being exchanged between *Alice* and *Bob*. Neural Network based cryptography develops a cryptosystem using separate neural networks for *Alice*, *Bob* and *Eve*, where the networks are trained to learn encryption, decryption and eavesdropping, (i.e. decryption without knowledge of the key) respectively. Generally, the networks for *Alice* and *Bob* share the secret key which is necessary for encryption and decryption (for symmetric key encryption),

whereas the adversary, *Eve* tries to obtain the plaintext from the ciphertext produced by the encryption function. The aim of the adversary is to reproduce the plaintext with as much accuracy as possible whereas the aim of encrpytion-decryption network is to minimize the difference between the output produced at the end of the decryption function and the original plaintext and at the same time maximize the error between the output produced by *Eve* and the original plaintext. The competition between the two networks ensures that the network is not over-trained (i.e. all bits of $Bob's$ output are same as the bits in the input plaintext and the bits in $Eve's$ output are completely different from the input plaintext in which case *Eve* can simply reverse his/her bits and obtain the original input plaintext).

The aim of the cryptosystem built using neural networks is to specifically ensure confidentiality and secrecy properties, among the various information security objectives. Besides, machine learning has also been used in cryptography for generating and establishing secret keys as well as for preventing the corresponding attacks based on predictibility of keys.

Here we summarize 3 approaches which have been adopted for implementing cryptographic schemes specifically encryption and decryption transformations using neural networks.

5.2 Adversarial Neural Cryptography

Basically, in this paper, neural networks are proposed for encryption and decryption networks as well as the adversary network which learn to fulfill their respective goals. The encryption-decryption network learns its own cryptographic scheme including the encryption and decryption transformations without any prior knowledge or any explicit supervision to train for some specific algorithm. Thus, the algorithm developed by the neural-based cryptosystem is completely architecture-specific and remains opaque to even the parties implementing the architecture. Besides, the net-

work parameters change with each training process due to random initializations, thus preventing a large class of attacks possible with other encryption schemes. Another work on automatic synthesis of cryptosystems is ZooCrypt[].

5.2.1 Training and Formulation

Consider P to be the input plaintext, K the secret key shared between *Alice* and *Bob*, C the ciphertext produced by $Alice's$ network, P_{Bob} the output produced by Bob's network and P_{Eve} the output produced by $Eve's$ network. Let the parameters for $Alice's$, $Bob's$ and $Eve's$ networks, be θ_A, θ_B and θ_E respectively. The values of C,P_{Bob} and P_{Eve} are constrained between $(-1, 1)$, while the parameters are real-valued. The aim of the $Alice - Bob$ architecture is to prevent the most optimal *Eve* from being able to predict more than half-of the bits of the input plaintext, so it is important for the *Eve* architecture to also become better so that $Alice - Bob$ network can defeat the most optimal *Eve*. For this reason, the paper proposes training the $Alice - Bob$ architecture for one mini-batch followed by training the *Eve* architecture for 2 mini-batches. Further the objective functions for the two networks are defined as:

Let us define, $d(P, P') = ||P - P'||_1$ i.e. the L1-norm of P and P'.

$C = A(\theta_A, P, K)$, where A represents $Alice's$ network thus, C represents the output of Alice's network for parameters θ_A and inputs P and K.

Then, the loss function for *Eve* and *Bob* are defined as:

$L_E(\theta_A, \theta_E) = \mathbb{E}(d(P, E(\theta_E, C)))$, where $E(.)$ represents the output for Eve's network for the given arguments and $\mathbb{E}$ specifies an expectation over all values obtained for the minibatch.

The optimal Eve is given by,

$$O_E(\theta_A) = argmin_{\theta_E}(L_E(\theta_A, \theta_E))$$

$L_B(\theta_A, \theta_B) = \mathbb{E}(d(P, B(\theta_B, C, K)))$, where $B(.)$ represents the output for Bob's network for the specified arguments

The overall loss function for $Alice - Bob$ architecture is given as (minimizing loss of Bob and maximizing loss of Eve for optimal Eve):

$$L_{AB}(\theta_A, \theta_B) = L_B(\theta_A, \theta_B) - L_E(\theta_A, O_E(\theta_A))$$

or, $L_{AB}(\theta_A, \theta_B) = L_B(\theta_A, \theta_B) - \frac{((\frac{N1}{2}) - L_E(\theta_A, O_E(\theta_A)))^2}{(\frac{N1}{2})^2}$, where $N1$ is the input plaintext length (Here it ensures that for Eve almost half bits are wrong)

$$(O_A, O_B) = argmin_{(\theta_A, \theta_B)}(L_{AB}(\theta_A, \theta_B))$$

The architectures proposed in the paper for $Alice$, Bob and Eve are given in Figure 5.1. The parameters for the networks are initialized randomly possibly using glorot xavier initialization[], then the training procedure is started. First the Alice-Bob architecture is trained with the loss function being $L_{AB}(.)$ as defined above, for one minibatch where instead of simply trying to maximize the error of Eve's reconstruction, the aim is to minimize the mutual information between P and $Eve's$ output i.e. trying to make his/her output as random as possible. Then, the Eve's network is trained with the loss function being $L_B(.)$ as defined above, for two minibatches.

5.2.2 Experimentation and Results

5.2.3 Further modifications

The paper [], further explores modifications to the network defined by [] in their architecture as explained in 5.2.1. First they propose changes in the learning rates and batch sizes as well as other hyperparameters such as no. of layers in the network and stride length. Then they obtain results by changing the size of the input secret keys with a small change in the architecture, the dense layers now have size $m + n$; where m is the new secret key length and n is the input plaintext length and $m < n$; in all the 3 architectures wherever present and also a final Fully-Connected layer is added

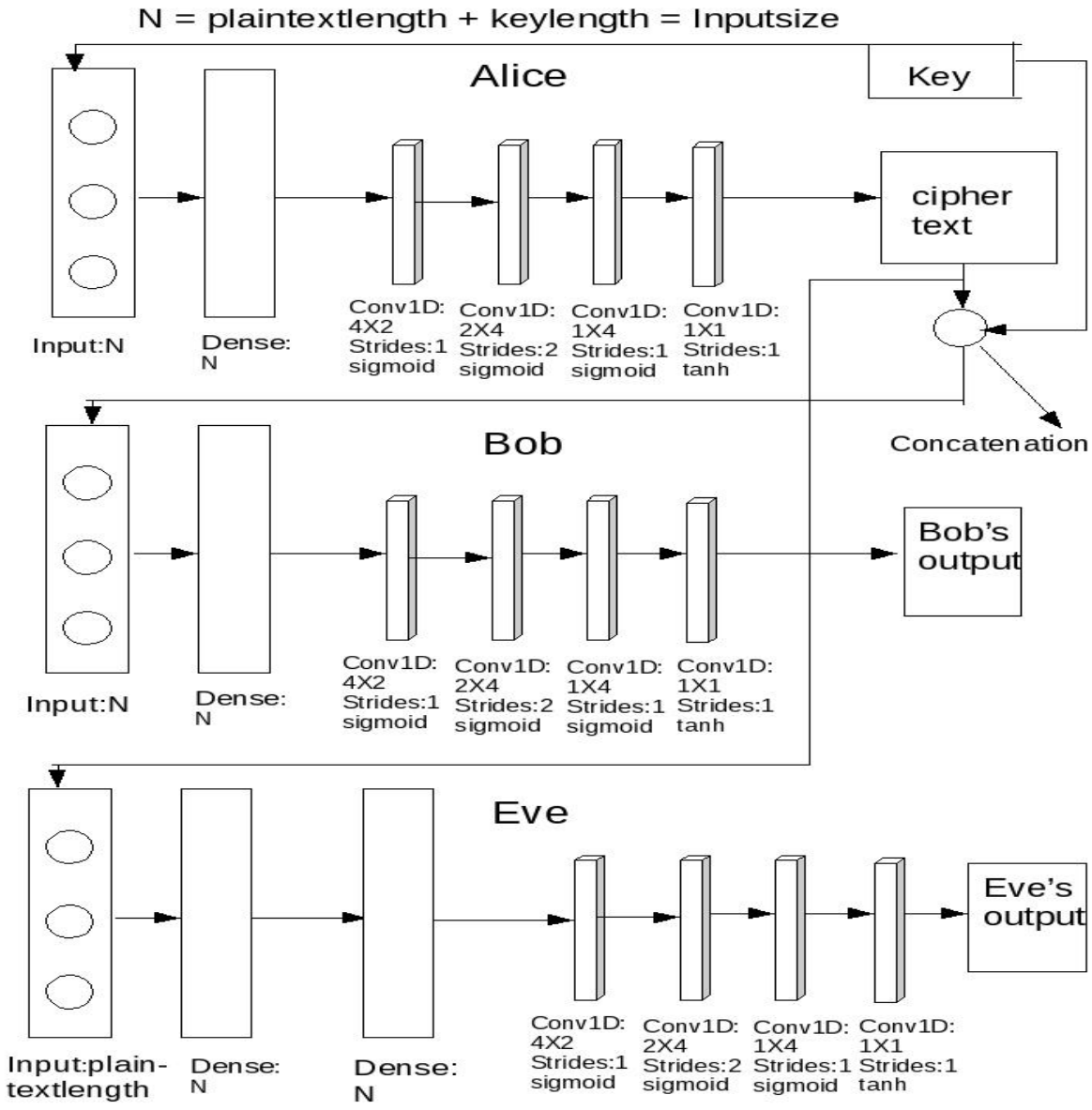

Figure 5.1 The architecture's used for *Alice*, *Bob* and *Eve*.[]

to each of the 3 architectures to produce outputs of the required shape i.e. the output of the final convolution layer is now $\frac{m+n}{2}$, so the FC layer is of size $\frac{m+n}{2} \times n$ to produce n outputs for each of the 3 architectures. The results obtained in the paper indicate that shorter keys lead to decrease in performance of the neural network architecture but still do perform resonably for $(m, n) = (12, 16)$.

Finally, the paper proposes a threshold cryptosystem which uses the concept of secret sharing so that the secret p is encoded as the constant coefficient of a $t - 1$ degree polynomial and the other coefficients are randomly initialized. This is done by considering $plaintext_length$ no. of polynomials where each polynomial would correspond to 1 bit of the secret message p. Now, n points on the curves of these polynomials are encoded as plaintext messages(with the xcoordinates of the points being same for every polynomial), which is then passed one each through an ensemble of the proposed neural network architectures with total n networks as specified earlier where each neural network gives an output for the corresponding output of Bob so that now again using the obtained output (representing new xcoordinates) and the y values of the coordinates of original points, new n points are determined which are then used to approximate polynomials passing through these points, and the constant coefficients of these polynomials are used to reconstruct the original message. As close the obtained output of the networks would be to the input of the networks respectively, so would the reconstructed points and the original points be close to each other thus, approximating the original polynomials used to form the input plaintext messages and thus, leading to more accurate reconstruction of original message p by Bob. Eve would also attempt reconstructing the message p but since his/her points are far from the original points on the polynomials thus, reconstruction of p by Eve is inaccurate. The threshold cryptosystem has been implemented as well using Tensorflow. The code is available here.