1. What are hyper parameters?

- In addition to the weight parameters defined in our neural network, learning algorithms also require a couple of additional parameters to carry out the training process. Eg:*learning rate., batch size*

1. The Limits of Traditional Computer Programs
2. How machine learning/Deep Learning can solve the problems what traditional programs can not do(or) Mechanics of Machine Learning
3. Outline the feed forward network with the neat diagram
   (or)
   Draw and explain about Multi layer perceptron

   (OR)

   Explain perceptron and how to solve and/or logic gates with perceptron
4. Discuss gradient descent with sigmoidal neurons
5. Explain delta rule for training linear neurons
6. How can you compare artificial neuron to biological neuron?
7. Expalin the working of perceptron
8. Expalin about different activation functions(or) neurons.
9. Define linear neuron and find limitations of it in detail
10. Summarize back propagation algorithm in reducing the error.

Unit 2

Discuss the importance of convolution neural networks to overcome the short comings of feature selection.

Outline the importance of max pooling with its variants in detail

Examine the function of convolutional layer with its characteristics

Explain the architecture of VGGNet, with the help of a neat diagram

Describe the full architecture of cnn

## Explain Image Preprocessing techniques used in cnn

Natural images are messy, and as a result, there are a number of preprocessing operations

that we can utilize in order to make training slightly easier. The first techniquethat is supported out of the box in TensorFlow is approximate per-image whitening.
The basic idea behind whitening is to zero-center every pixel in an image by subtracting out the mean and normalizing to unit 1 variance. This helps us correct for potential differences in dynamic range between images. In TensorFlow, we can achieve this using:

```
tf.image.per_image_whitening(image)
```

We also can expand our dataset artificially by randomly cropping the image, flipping the image, modifying saturation, modifying brightness, etc: Some exaples:

```
tf.random_crop(value, size, seed=None, name=None)
tf.image.random_flip_up_down(image, seed=None)
tf.image.random_flip_left_right(image, seed=None)
tf.image.transpose_image(image)
tf.image.random_brightness(image, max_delta, seed=None)
```

## Expalinabout  Batch Normalization

Training of feed-forward and convolutional neural networks can be accelerated using a technique called*batch normalization*

In the process of training the weights of the network, theoutput distribution of the neurons in the bottom layer begins to shift. The result ofthe changing distribution of outputs from the bottom layer means that the top layernot only has to learn how to make the appropriate predictions, but it also needs tosomehow modify itself to accommodate the shifts in incoming distribution. This significantlyslows down training, and the magnitude of the problem compounds themore layers we have in our networks.

Normalization of image inputs helps out the training process by making it morerobust to variations. Batch normalization takes this a step further by normalizinginputs to every layer in our neural network. Specifically, we modify the architectureof our network to include operations that:
1. Grab the vector of logits incoming to a layer before they pass through the nonlinearity
2. Normalize each component of the vector of logits across all examples of the minibatch
by subtracting the mean and dividing by the standard deviation (we keep
track of the moments using an exponentially weighted moving average)
3. Given normalized inputs $\hat{\mathbf{x}}$, use an affine transform to restore representational
power with two vectors of (trainable) parameters: $\gamma\hat{\mathbf{x}} + \beta$

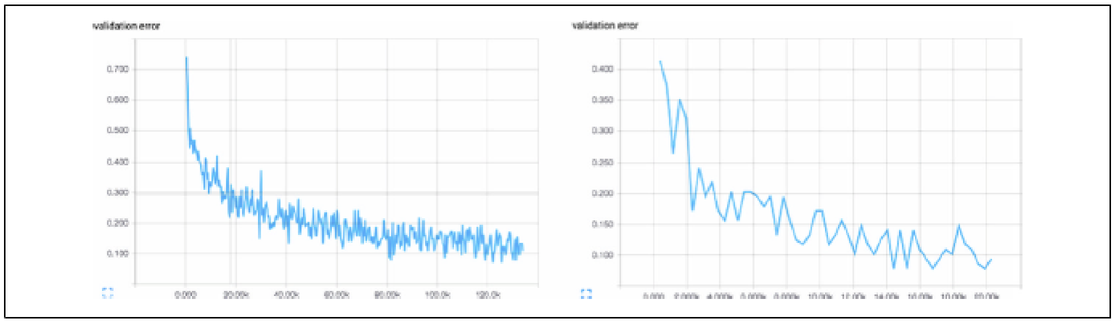


*Figure 5-17. Training a convolutional network without batch normalization (left) versus with batch normalization (right). Batch normalization vastly accelerates the training process.*

Without batch normalization, cracking the 90% accuracy threshold requires over80,000 minibatches. On the other hand, with batch normalization, crossing the samethreshold only requires slightly over 14,000 minibatches.

Elaborate on the principal component analysis for dimensionality reduction
Explain the architecture of auto encoders
Elaborate on the auto encoder for dimensionality reduction
Describe sparsity in auto encoders to improve its performance
Elaborate Word2Vec framework to generate word embeddings
Explain one hot encoding

Unit 3

How to compute gradient in recurrent neural networks? Relate with an example
Explain the different models of Re current Neural Networks

ExplainEncoder-Decoder sequence to sequence architectures?
What is a recursive neural network? Differntaite recurrent and recursive network

Explain deep recurrent network

Discuss the challenge of long term dependencies
Illustrate with an example how recurrent neural networks can be considered as a directed graphical model?
Elaborate on echo state networks

What is LSTM? Explain with a neat block diagram.

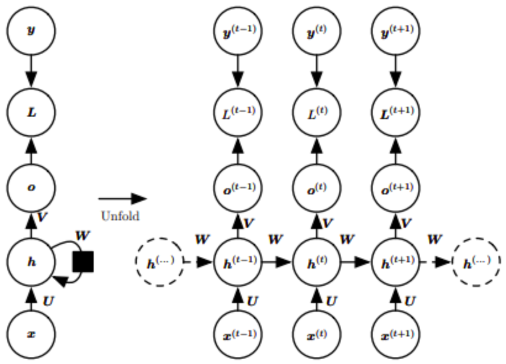
Assess the challenges of long term dependencies in RNNS and suggest some remedial methods to avoid those challenges

Explain briefly about Bidirectional RNNS

Create an RNN architecture for summarizing a sequence and produce a fixed- size representation
What are leaky units?

How to compute gradient in recurrent neural networks? Relate with an example

The nodes of our computational graph include the parameters U, V , W, b and c as well as the sequence of nodes indexed by t for x (t) , h (t) , o (t) and L (t) . For each node N we need to compute the gradient ∇NL recursively, based on the gradient computed at nodes that follow it in the graph. We start the recursion with the nodes immediately preceding the final loss:
Considering an example of a recurrent network that maps an input sequence to an output sequence of the same length:



Here we assume the hyperbolic tangent activation function. Also, the figure does not specify exactly what form the output and loss function take. Here we assume that the output is discrete, as if the RNN is used to predict words or characters. . Forward propagation begins with a specification of the initial state h (0). Then, for each time step from t = 1 to t = τ , we apply the following update equations:

$$\begin{aligned}
\boldsymbol{a}^{(t)} &= \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)}, \\
\boldsymbol{h}^{(t)} &= \tanh(\boldsymbol{a}^{(t)}), \\
\boldsymbol{o}^{(t)} &= \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)},
\end{aligned}$$

$$\hat{\boldsymbol{y}}^{(t)} = \operatorname{softmax}(\boldsymbol{o}^{(t)}),$$

where the parameters are the bias vectors b and c along with the weight matrices U, V and W, respectively, for input-to-hidden, hidden-to-output and hidden to-hidden connections. The total loss for a given sequence of x values paired with a sequence of y values would then be just the sum of the losses over all the time steps. For example, if L (t) is the negative log-likelihood of y (t) given x (1) , . . . , x (t) , then

$$
\begin{aligned}
&L\left(\{\boldsymbol{x}^{(1)},\ldots,\boldsymbol{x}^{(\tau)}\},\{\boldsymbol{y}^{(1)},\ldots,\boldsymbol{y}^{(\tau)}\}\right)\\
&=\sum_t L^{(t)}\\
&=-\sum_t \log p_{\text{model}}\left(y^{(t)} \mid \{\boldsymbol{x}^{(1)},\ldots,\boldsymbol{x}^{(t)}\}\right),
\end{aligned}
$$

where pmodel y (t) | {x (1) , . . . , x (t)}  is given by reading the entry for y (t) from the model's output vector yˆ (t)
The nodes of our computational graph include the parameters U, V , W, b and c as well as the sequence of nodes indexed by t for x (t) , h (t) , o (t) and L (t) . For each node N we need to compute the gradient ∇NL recursively, based on the gradient computed at nodes that follow it in the graph. We start the recursion with the nodes immediately preceding the final loss:

$$\frac{\partial L}{\partial L^{(t)}}=1.$$

In this derivation we assume that the outputs o (t) are used as the argument to the softmax function to obtain the vector yˆ of probabilities over the output. We also assume that the loss is the negative log-likelihood of the true target y (t) given the input so far. The gradient ∇o(t)L on the outputs at time step t, for all i, t, is as follows:

$$(\nabla_{\boldsymbol{o}^{(t)}}L)_i=\frac{\partial L}{\partial o_i^{(t)}}=\frac{\partial L}{\partial L^{(t)}}\frac{\partial L^{(t)}}{\partial o_i^{(t)}}=\hat{y}_i^{(t)}-\mathbf{1}_{i,y^{(t)}}.$$

We work our way backward, starting from the end of the sequence. At the final time step τ , h (τ) only has o (τ) as a descendent, so its gradient is simple:

$$\nabla_{\boldsymbol{h}^{(\tau)}}L=\boldsymbol{V}^{\top}\nabla_{\boldsymbol{o}^{(\tau)}}L.$$

We can then iterate backward in time to back-propagate gradients through time, from t = τ − 1 down to t = 1, noting that h (t) (for t < τ ) has as descendents both o (t) and h (t+1). Its gradient is thus given by

$$
\begin{aligned}
\nabla_{\boldsymbol{h}^{(t)}}L&=\left(\frac{\partial \boldsymbol{h}^{(t+1)}}{\partial \boldsymbol{h}^{(t)}}\right)^{\top}(\nabla_{\boldsymbol{h}^{(t+1)}}L)+\left(\frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{h}^{(t)}}\right)^{\top}(\nabla_{\boldsymbol{o}^{(t)}}L)\\
&=\boldsymbol{W}^{\top}(\nabla_{\boldsymbol{h}^{(t+1)}}L)\,\text{diag}\left(1-\left(\boldsymbol{h}^{(t+1)}\right)^2\right)+\boldsymbol{V}^{\top}(\nabla_{\boldsymbol{o}^{(t)}}L),
\end{aligned}
$$

where diag $(1 – (h^{(t+1)})^2)$ indicates the diagonal matrix containing the elements $1 − (h_i(t+1))^2$.  Once the gradients on the internal nodes of the computational graph are obtained, we can obtain the gradients on the parameter nodes.
the gradient on the remaining parameters is given by

$$
\begin{aligned}
\nabla_{\boldsymbol{c}}L&=\sum_t\left(\frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{c}}\right)^{\top}\nabla_{\boldsymbol{o}^{(t)}}L=\sum_t\nabla_{\boldsymbol{o}^{(t)}}L,\\
\nabla_{\boldsymbol{b}}L&=\sum_t\left(\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{b}^{(t)}}\right)^{\top}\nabla_{\boldsymbol{h}^{(t)}}L=\sum_t\text{diag}\left(1-\left(\boldsymbol{h}^{(t)}\right)^2\right)\nabla_{\boldsymbol{h}^{(t)}}L,\\
\nabla_{\boldsymbol{V}}L&=\sum_t\sum_i\left(\frac{\partial L}{\partial o_i^{(t)}}\right)\nabla_{\boldsymbol{V}}o_i^{(t)}=\sum_t(\nabla_{\boldsymbol{o}^{(t)}}L)\,\boldsymbol{h}^{(t)^{\top}},
\end{aligned}
$$

$$\begin{aligned}
\nabla_{\boldsymbol{W}} L &= \sum_t \sum_i \left( \frac{\partial L}{\partial h_i^{(t)}} \right) \nabla_{\boldsymbol{W}^{(t)}} h_i^{(t)} \\
&= \sum_t \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t)}\right)^2\right) (\nabla_{\boldsymbol{h}^{(t)}} L)\, \boldsymbol{h}^{(t-1)\top}, \\
\nabla_{\boldsymbol{U}} L &= \sum_t \sum_i \left( \frac{\partial L}{\partial h_i^{(t)}} \right) \nabla_{\boldsymbol{U}^{(t)}} h_i^{(t)} \\
&= \sum_t \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t)}\right)^2\right) (\nabla_{\boldsymbol{h}^{(t)}} L)\, \boldsymbol{x}^{(t)\top},
\end{aligned}$$

Discuss the challenge of long term dependencies.

The basic problem is that gradients propagated over many stages tend to either vanish (most of the time) or explode (rarely, but with much damage to the optimization). Even if we assume that the parameters are such that the recurrent network is stable (can store memories, with gradients not exploding), the difficulty with long-term dependencies arises from the exponentially smaller weights given to long-term interactions (involving the multiplication of many Jacobians) compared to short-term ones.

Suppose a computational graph consists of repeatedly multiplying by a matrix W. After t steps this is equivalent to multiplying by Wt.

• Suppose W has an eigendecomposition W=Vdiag(λ)$V^{-1}$ . • In this case it is straightforward to see that

$$W_t = (V\mathrm{diag}(\lambda)V^{-1})^t = V\mathrm{diag}(\lambda)^t\, V^{-1}$$

• Any eigenvalues λi that are not near an absolute value of 1 will either explode if they are greater than 1 in magnitude and vanish if they are less than 1 in magnitude. Vanishing gradients make it difficult to know which direction the parameters should move to improve cost.

• Recurrent networks involve the composition of the same function multiple times, once per time step. These compositions can result in extremely nonlinear behavior

Composing many nonlinear functions: tanh here h has 100 dimensions mapped to a single dimension Most of the space it has a small derivative and highly nonlinear elsewhere.

Illustrate with an example how recurrent neural networks can be considered as a directed graphical model?

- we train the RNN to estimate the conditional distribution of the next sequence element *y*(*t*) given the past inputs.
- This may mean that we maximize the log-likelihood

$$\log p(\boldsymbol{y}^{(t)} \mid \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t)}),$$

- if the model includes connections from the output at one time step to the next time step,

$$\log p(\boldsymbol{y}^{(t)} \mid \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t)}, \boldsymbol{y}^{(1)}, \ldots, \boldsymbol{y}^{(t-1)}).$$

- Decomposing the joint probability over the sequence of *y* values as a series of one-step probabilistic predictions is one way to capture the full joint distribution across the whole sequence
- As a simple example, let us consider the case where the RNN models only a sequence of scalar random variables Y = {y(1), . . . , y(τ)}, with no additional inputs x.
- The input at time step *t* is simply the output at time step *t* −1. The RNN then defines a directed graphical model over the y variables.

- We parametrize the joint distribution of these observations using the chain rule for conditional probabilities:

$$P(\mathbb{Y}) = P(\mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(\tau)}) = \prod_{t=1}^{\tau} P(\mathbf{y}^{(t)} \mid \mathbf{y}^{(t-1)}, \mathbf{y}^{(t-2)}, \ldots, \mathbf{y}^{(1)})$$

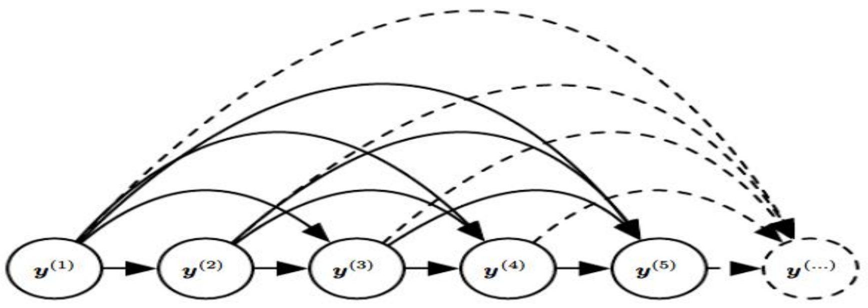
- Hence the negative log-likelihood of a set of values $\{y(1), \ldots, y(\tau)\}$ according to such a model is

$$L = \sum_t L^{(t)}$$

- Where

$$L^{(t)} = -\log P(\mathrm{y}^{(t)} = y^{(t)} \mid y^{(t-1)}, y^{(t-2)}, \ldots, y^{(1)}).$$

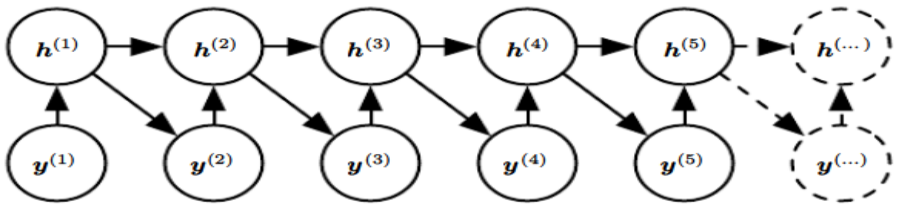
- One way to interpret an RNN as a graphical model is to view the RNN as defining a graphical model whose structure is the complete graph, able to represent direct dependencies between any pair of y values.



- Here, every past observation $y(i)$ may influence the conditional distribution of some $y(t)$ (for $t > i$), given the previous values.
- Parametrizing the graphical model directly according to this graph might be very inefficient, with an ever growing number of inputs and parameters for each element of the sequence.

Introducing the state variable

- Incorporating the $h(t)$ nodes in the graphical model decouples the past and the future, acting as an intermediate quantity between them.
- A variable $y(i)$ in the distant past may influence a variable $y(t)$ via its effect on $h$.
- Every stage in the sequence (for $h(t)$ and $y(t)$ ) involves the same structure (the same number of inputs for each node) and can share the same parameters with the other stages.



Elaborate on echo state networks.

The recurrent weights mapping from h (t−1) to h (t) and the input weights mapping from x (t) to h (t) are some of the most difficult parameters to learn in a recurrent network. One proposed approach to avoiding this difficulty is to set the recurrent weights such that the recurrent hidden units do a good job of capturing the history of past inputs, and only learn the output weights. This is the idea that was independently proposed for echo state networks, or ESNs. ESNs are termed reservoir computing to denote the fact that the hidden units form a reservoir of temporal features that may capture different aspects of the history of inputs and has the following features

- the weights between the input -the hidden layer ( the 'reservoir') : *Win* and also the weights of the 'reservoir': *Wr* are randomly assigned and not trainable
- the weights of the output neurons (the 'readout' layer) are trainable and can be learned so that the network can reproduce specific temporal patterns
- the hidden layer (or the 'reservoir') is very sparsely connected (typically < 10% connectivity)
- the reservoir architecture creates a recurrent non linear embedding (H on the image below) of the input which can be then connected to the desired output and these final weights will be trainable
- it is possible to connect the embedding to a different predictive model (a trainable NN or a ridge repressor/SVM for classification problems)

One way to think about these reservoir computing recurrent networks is that they are similar to kernel machines: they map an arbitrary length sequence (the history of inputs up to time t) into a fixed-length vector (the recurrent state h (t)), on which a linear predictor (typically a linear regression) can be applied to solve the problem of interest.

UNIT 4

1. Explain concept of Attention mechanism
2. Analyze the architecture of Visual attention wrt human retina system.
2. Discuss the application of the visual attention approach for image captioning
3. Explain the working of Neural Turing machine with a neat diagram
4. What is GAN? Explain the process of generating image data with GAN
5.Discuss the limitations of Neural Networks