CS 217: Artificial Intelligence and Machine Learning **Jan-Apr 2024**

Lecture 12: Neural Networks Training Algorithm

Lecturer: Prof. Swaprava Nath *Scribe(s): SG23, SG24*

Disclaimer: *These notes aggregate content from several texts and have not been subjected to the usual scrutiny deserved by formal publications. If you find errors, please bring to the notice of the Instructor.*

In this lecture, we study the algorithm used to train feed forward neural networks. It consists of two major steps:

- Forward propagation *and*
- Backward propagation.

12.1 Forward Propagation

Forward Propagation, also known as **forward feed**, is a fundamental step in training neural networks, particularly in supervised learning tasks such as classification or regression. It involves the process of passing input data through the network to obtain output predictions.

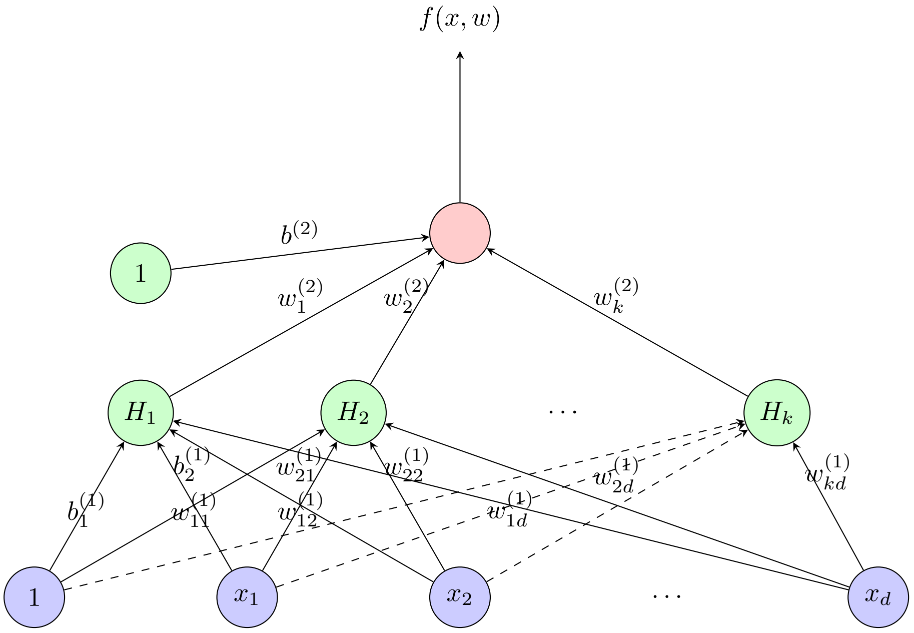


Figure 12.1: A two-layer Neural Network

In this diagram H_1, H_2, ..., H_k denote the hidden nodes, f(x, w) denotes the predicted probability for the corresponding input x which is $[x_1, x_2, ..., x_d]^t$.

Neural Networks are a way to approximate functions[1] given the presence of labelled data. Ultimately, our purpose is to create a model that is nothing but our neural network that can predict the output of an unseen input. So, basically. we represent the value in the i^{th} node and the j^{th} layer as x_{ij}. So the vector of values for the j^{th} layer is given as

$$\mathbf{X}_j = \begin{bmatrix} x_{1j} \\ x_{2j} \\ \vdots \\ x_{ij} \\ \vdots \\ x_{nj} \end{bmatrix}$$

where n is the number of nodes in the j^{th} layer. We construct a list of such vectors for all layers of the nodes:

$$Layer\_List \; = \; [X_1, X_2, \ldots, X_m]$$

We know the edges in the neural network graph represent weights if the number of nodes in the i^{th} and the j^{th} layer be $n_i \, and \, n_j$ respectively, so we need n_i weights to link the relationship between one node in the j^{th} layer to each node of the i^{th} layer. With n_j such sets of weights, we can represent the weights between these two layers by a $n_j \times n_i$ matrix.

The weights between the i^{th} and j^{th} layers can be represented as a matrix:

$$W_{ij} = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1n_i} \\ w_{21} & w_{22} & \cdots & w_{2n_i} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n_j1} & w_{n_j2} & \cdots & w_{n_jn_i} \end{bmatrix}$$

We can similarly construct a weights list for weights between the layers.

$$Weights\_List \; = \; [W_{12}, W_{23}, \ldots, W_{m-1m}]$$

What is forward pass now?

In the Layer_List, we know X_1, which is our input, and our job is to find ${X_m}^2$, our output.

So, to do this, we iteratively apply the relation:

$$X_j \; = \; W_{ij} \cdot X_i$$

Forward_Pass($Weights\_List$**,**X_1**=Function_Input):**

[1] The statement that neural networks are universal function approximators is based on a mathematical result known as the universal approximation theorem. This theorem states that certain types of neural networks can approximate any continuous function on a compact subset of the input space to any desired degree of accuracy, given enough hidden units [hornik 1989 multilayer, cybenko 1989 approximation]. This implies that neural networks have the potential to learn any complex pattern or relationship from data as long as the underlying function is continuous and bounded.

However, the universal approximation theorem does not guarantee that neural networks can always find the optimal weights to achieve the desired approximation. It does not specify how many hidden units are required for a given problem. It also does not account for the generalization ability of neural networks, which is how well they perform on unseen data. Therefore, the statement should not be interpreted as a claim that neural networks can solve any problem perfectly but rather as a theoretical possibility that motivates their use in various domains [goodfellow2016deep, schmidhuber2015deep, lecun2015deep].

[2] We define our loss function over this X_m

```
for $i$ in range(2 to m) then
    $X_i = W_{i-1i} \cdot X_{i-1}$
return $X_m$
```

Hidden Nodes:

$$H_i = \mathbf{g}([w_i^{(1)}]^t x + b_i^{(1)})$$

Predicted Probability:

$$f(x, w) = \mathbf{g}([w^{(2)}]^t H + b^{(2)})$$

Cross Entropy:

$$l_i = -y_i log(f(x_i, w)) - (1 - y_i) log(1 - f(x_i, w))$$

12.1.1 Mini Batch Gradient Descent

Goal: To learn weights and biases (w, b) that minimize the loss

Algorithm 1 Mini Batch Gradient Descent

```
randomly initialize $\theta(w, b)$
while stopping condition is not met do
    Randomly pick a batch of examples $(\mathbf{X_i}, y_i)_{i \in \mathbf{B}}$ where $\mathbf{B} \subset \{1, 2, \ldots, n\}$ , $and |\mathbf{B}| = T < n$
    Compute the gradient of the loss function
    Update $\theta \leftarrow \theta$ - $\eta \nabla_\theta l$, where $\eta$ is the learning rate
return $\theta$
```

12.1.2 Efficient Computation of Gradients

1) Compute $\left.\frac{\partial l}{\partial H}\right|_\theta$ for every H in the **NN**.

2) Compute $\frac{\partial l}{\partial w} = \frac{\partial l}{\partial H} \cdot \frac{\partial H}{\partial w}$, where $H = \mathbf{g}(\text{wz} + ...)$ and z are previous nodes $\Rightarrow \frac{\partial H}{\partial w} = \mathbf{g}^1.z.$

3) $\frac{\partial l}{\partial H} = \sum_{v \in \Gamma(H)} \frac{\partial l}{\partial v} \cdot \frac{\partial v}{\partial H}$, where $\Gamma(H) = [v_1, v_2, ...]$ denotes the next layer of H and v $= \mathbf{g}(\widetilde{w} H + ...)$.

12.2 Back Propagation

This is a two-stage algorithm. It consists of the following steps

1) Forward Pass

We calculate the value of each node given an input (x_i, y_i) and parameters θ. For the j^{th} node of the k^{th} layer of the neural network, we use the following equation

$$u_j = (w_{j1}^{(k)} z_1 + w_{j2}^{(k)} z_2 + \cdots + b_j^{(k)})$$

2) Backward Pass

We use the following algorithm, with base case $\frac{\partial l}{\partial l} = 1$

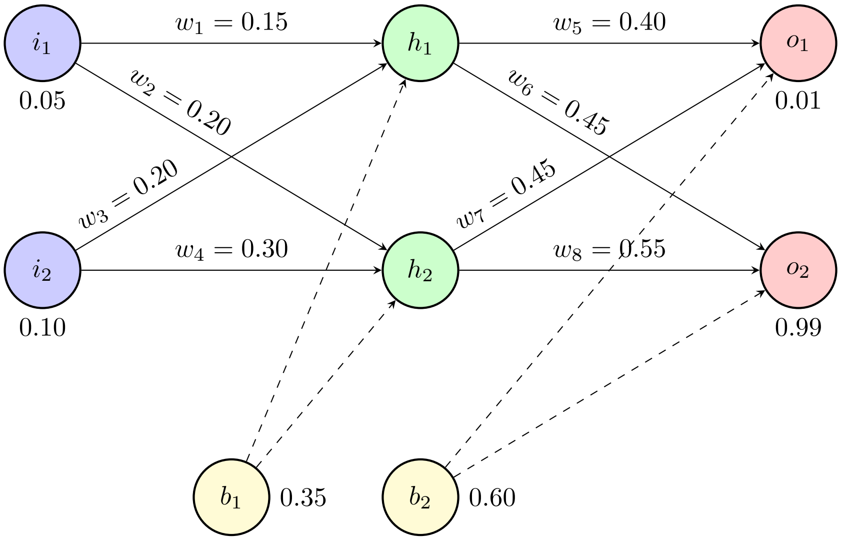
Algorithm 2 Backward Pass

```
for each $u$ in a given layer do
    for each $v$ in $\Gamma(u)$ do
        Use already computed $\frac{\partial l}{\partial v}$ (assumed to be known by induction)
        Compute $\frac{\partial v}{\partial u}|_{x_i, y_i, \theta}$
        Compute $\frac{\partial l}{\partial v} \cdot \frac{\partial v}{\partial u}$
    Get $\frac{\partial l}{\partial u}$, compute $\frac{\partial l}{\partial w} = \frac{\partial l}{\partial u} \cdot \frac{\partial u}{\partial w}$ and $\frac{\partial l}{\partial b}$
```

The forward and backward pass together make up one iteration of gradient descent. We update the parameters θ and move on to the next iteration.

12.2.1 Example

Let us illustrate back propagation with an example. Consider the following neural network with the indicated initial weights, biases and training inputs with the true outputs.



We will now use back propagation to optimize the weights so that the neural network can make correct predictions. We will consider a training dataset of only the two points shown in the figure.

1) Forward Pass

Let us first see what our neural network currently predicts (with the randomly chosen weights and biases shown above). We do this using the sigmoid activation function in both the hidden layer nodes and the output layer nodes.

$$net_{h1} = w_1 * i_1 + w_3 * i_2 + b_1 * 1$$

$$net_{h1} = 0.15 * 0.05 + 0.2 * 0.1 + 0.35 * 1 = 0.3775$$

Applying the sigmoid function,

$$out_{h1} = \sigma(net_{h1}) = 0.593269992$$

Repeating the same process for h_2 we get

$$out_{h2} = 0.596884378$$

Using the same method for the output layer nodes with the output from the hidden layer neurons as inputs,

$$net_{01} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$net_{o1} = 0.4 * 0.593269992 + 0.45 * 0.596884378 + 0.6 * 1 = 1.105905967$$

$$out_{o1} = \sigma(1.105905967) = 0.75136507$$

$$out_{o2} = 0.772928465$$

Calculating the total error

We now calculate the error for each output neuron using the squared error function and sum them to get the total error.

$$E_{total} = \Sigma\frac{1}{2}(target - output)^2$$

Using this formula,

$$E_{o1} = 0.274811083, E_{o2} = 0.023560026$$

$$E_{total} = E_1 + E_2 = 0.298371109$$

2) Backward Pass

Our goal with backpropagation is to update each of the weights in the network so that they cause the actual output to be closer the target output, thereby minimizing the error for each output neuron and the network as a whole.

Output Layer

Consider w_5. We want to know how much a change in w_5 affects the total error, aka $\frac{\partial E_{total}}{\partial w_5}$.By applying the chain rule we know that:

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$

To compute the total error change with respect to the output, we have the formulas

$$E_{total} = \frac{1}{2}(target_{o1} - out_{o1})^2 + \frac{1}{2}(target_{o2} - out_{o2})^2$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = 2 * \frac{1}{2}(target_{o1} - out_{o1})^{2-1} * -1 + 0$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = -(target_{o1} - out_{o1}) = -(0.01 - 0.75136507) = 0.74136507$$

The partial derivative of the logistic function is the output multiplied by 1 minus the output.

$$out_{o1} = \frac{1}{1 + e^{-net_{o1}}}$$

$$\frac{\partial out_{o1}}{\partial net_{o1}} = out_{o1}(1 - out_{o1}) = 0.75136507(1 - 0.75136507) = 0.186815602$$

To calculate how much the total net input of o1 changes with respect to w_5,

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$\frac{\partial net_{o1}}{\partial w_5} = 1 * out_{h1} * w_5^{(1-1)} + 0 + 0 = out_{h1} = 0.593269992$$

To decrease the error, we then subtract this value from the current weight (optionally multiplied by some learning rate, η, which we'll set to 0.5).

$$w_5^+ = w_5 - \eta * \frac{\partial E_{total}}{\partial w_5} = 0.4 - 0.5 * 0.082167041 = 0.35891648$$

We can repeat this process to get the new weights w_6, w_7, and w_8:

$$w_6^+ = 0.408666186$$

$$w_7^+ = 0.511301270$$

$$w_8^+ = 0.561370121$$

We perform the actual updates in the neural network after we have the new weights leading into the hidden layer neurons (ie, we use the original weights, not the updated weights, when we continue the back propagation algorithm below).

Hidden Layer

Next, we continue the backwards pass by calculating new values for w_1, w_2, w_3, and w_4. We need to compute

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

We use a similar process as we did for the output layer, but slightly different to account for the fact that the output of each hidden layer neuron contributes to the output (and therefore error) of multiple output neurons. We know that out_{h1} affects both out_{o1} and out_{o2} therefore the $\frac{\partial E_{total}}{\partial out_{h1}}$needs to take into consideration its effect on the both output neurons.

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$

Starting with $\frac{\partial E_{o1}}{\partial out_{h1}}$

$$\frac{\partial E_{o1}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial out_{h1}}$$

We can calculate $\frac{\partial E_{o1}}{\partial net_{o1}}$ using values we calculated earlier.

$$\frac{\partial E_{o1}}{\partial net_{o1}} = \frac{\partial E_{o1}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} = 0.74136507 * 0.186815602 = 0.138498562$$

And $\frac{\partial net_{o1}}{\partial out_{h1}}$ is equal to w_5:

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$\frac{\partial net_{o1}}{\partial out_{h1}} = w_5 = 0.40$$

Plugging them in,

$$\frac{\partial E_{o1}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial out_{h1}} = 0.138498562 * 0.40 = 0.055399425$$

Following the same process for $\frac{\partial E_{o2}}{\partial out_{h1}}$, we get:

$$\frac{\partial E_{o2}}{\partial out_{h1}} = -0.019049119$$

Therefore

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}} = 0.055399425 + -0.019049119 = 0.036350306$$

Now that we have $\frac{\partial E_{total}}{\partial out_{h1}}$, we need to figure out $\frac{\partial out_{h1}}{\partial net_{h1}}$ and then $\frac{\partial net_{h1}}{\partial w}$for each weight.

$$out_{h1} = \frac{1}{1 + e^{-net_{h1}}}$$

$$\frac{\partial out_{h1}}{\partial net_{h1}} = out_{h1}(1 - out_{h1}) = 0.59326999(1 - 0.59326999) = 0.241300709$$

We calculate the partial derivative of the total net input to h_1 with respect to w_1 the same as we did for the output neuron.

$$net_{h1} = w_1 * i_1 + w_3 * i_2 + b_1 * 1$$

$$\frac{\partial net_{h1}}{\partial w_1} = i_1 = 0.05$$

Putting it all together,

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial w_1} = 0.036350306 * 0.241300709 * 0.05 = 0.000438568$$

We now have the updated weights. Repeating this process for several iterations further reduces the error (for example, $10,000$ iterations of this process results in an error of 0.000351085). This completes our illustration of back propagation.

12.3 Computational Graph

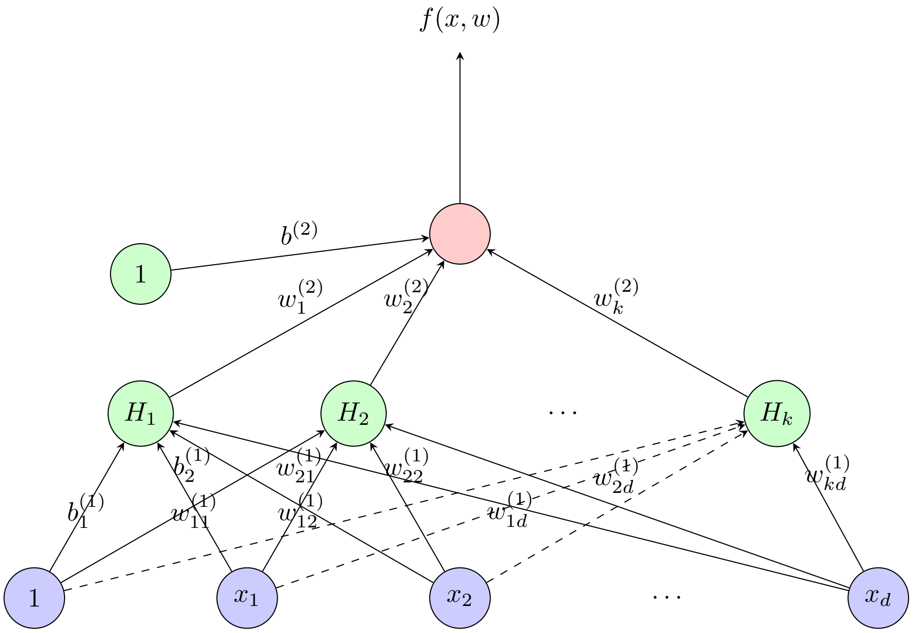
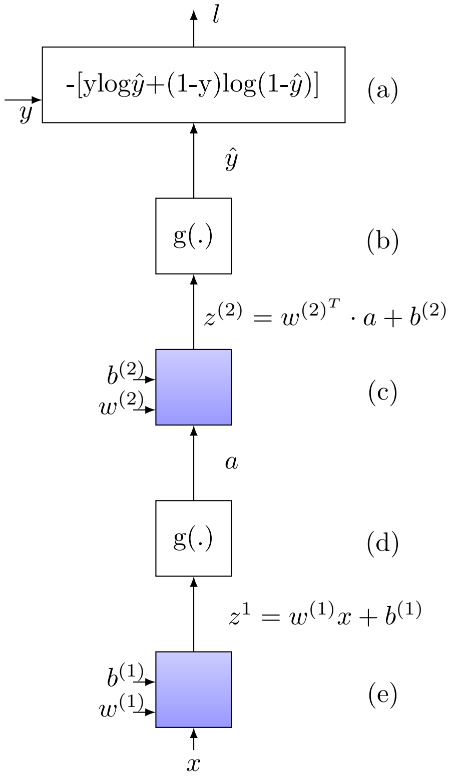


Figure 12.2: A two-layer Neural Network


l
-[ylog$\hat{y}$+(1-y)log(1-$\hat{y}$)]
(a)
y
$\hat{y}$
g(.)
(b)
$z^{(2)} = w^{(2)^T} \cdot a + b^{(2)}$
$b^{(2)}$
$w^{(2)}$
(c)
a
g(.)
(d)
$z^1 = w^{(1)}x + b^{(1)}$
$b^{(1)}$
$w^{(1)}$
(e)
x

The representation shown on the left here is the computational graph of the above two-layer Neural Network, with the loss function as $y \cdot log(\hat{y}) + (1-y)\cdot log(1-\hat{y})$.

$$\frac{\partial l}{\partial \hat{y}} = -\frac{\partial y \cdot log(\hat{y}) + (1-y)\cdot log(1-\hat{y})}{\partial \hat{y}} = -\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}} \tag{a}$$

$$\delta \hat{y} = \frac{\partial l}{\partial \hat{y}}\bigg|_{x,y,\theta}$$

$$\frac{\partial \hat{y}}{\partial z^{(2)}} = g'(z^{(2)}) \tag{b}$$

$$\delta z^{(2)} = \frac{\partial l}{\partial z^{(2)}} = \frac{\partial l}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z^{(2)}} = \delta\hat{y} \cdot \frac{\partial \hat{y}}{\partial z^{(2)}}$$

$$\frac{\partial z^{(2)}}{\partial w^{(2)}} = \frac{\partial (w^{(2)^T} a + b^{(2)})}{\partial w^{(2)}} = a \tag{c}$$

$$\delta w^{(2)} = \delta z^{(2)} \cdot \frac{\partial z^{(2)}}{\partial w^{(2)}}$$

$$\frac{\partial a^{(1)}}{\partial z^{(1)}} = g'(z^{(1)}) \tag{d}$$

$$\frac{\partial z^{(1)}}{\partial w^{(1)}} = \frac{\partial (w^{(1)}x + b^{(1)})}{\partial w^{(1)}} = x \implies \delta w^{(1)} = x \cdot \delta z^{(1)} \tag{e}$$

12.4 Regularization

12.4.1 L2 Regularization

$$l_{Reg} = l_{ce} + \lambda \|w\|_2^2$$

Here l_{ce} is the Cross-Entropy loss and $\|w\|_2^2 = \Sigma_{i,j,k}(w_{ij}^{(k)})^2$. This is the same L2 regularization we have seen so far.

12.4.2 Dropout Regularization

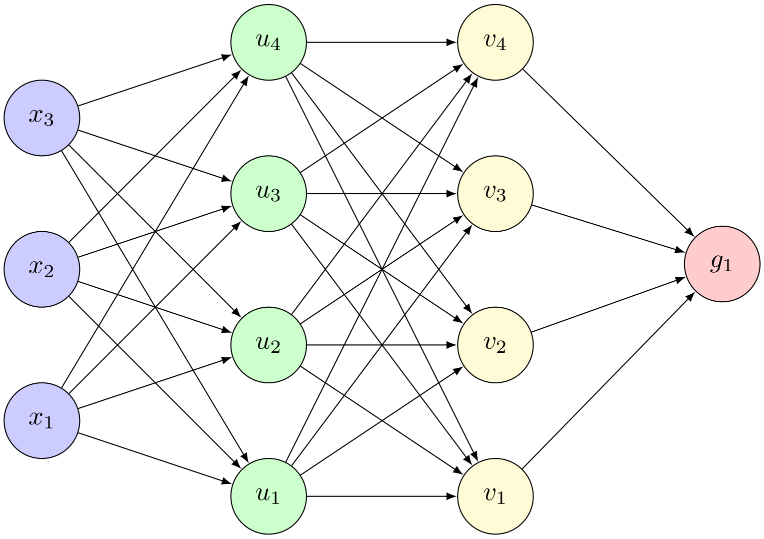


Figure 12.3: A fully connected NN with 2 hidden layers before dropout

During training, keep a neuron active with some probability p (hyperparameter), delete otherwise. For each node we do this process independently. We get a much sparser NN then where Back-Propagation calculations become simpler and model complexity is lowered. At test time, we use the entire network with the expected weights.

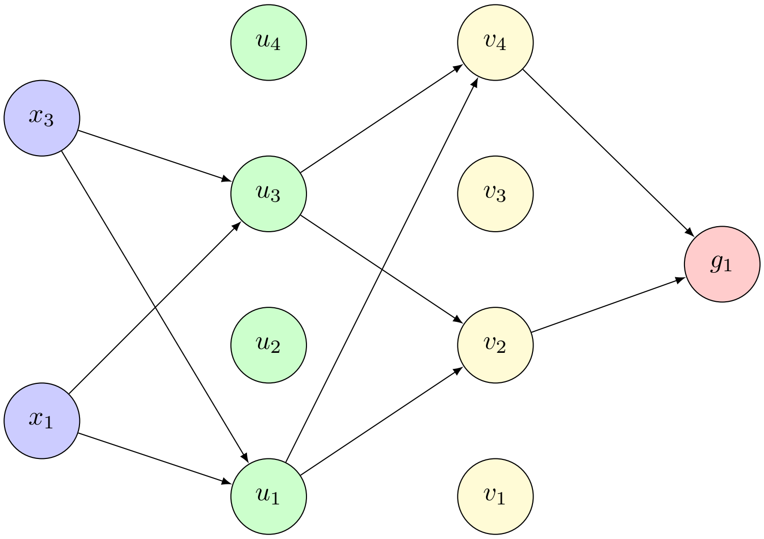


Figure 12.4: The same NN after dropping some nodes for a specific (x_i, y_i)

12.4.3 Early Stopping

Stopping when performance on validation set stops improving.

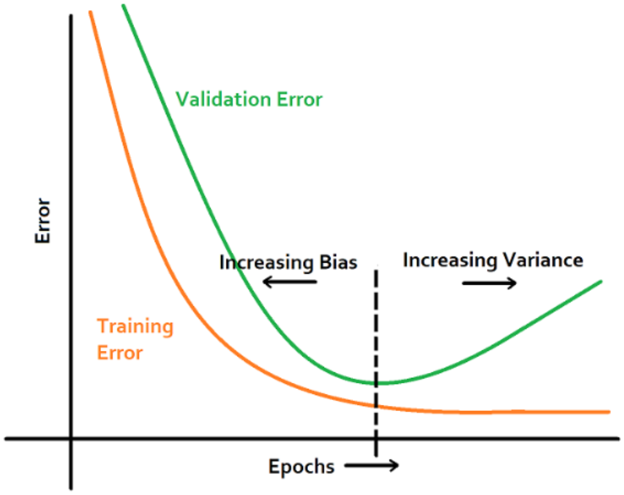
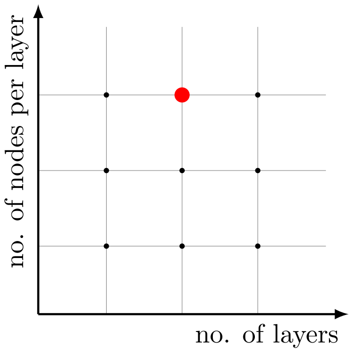


Figure 12.5: Caption



Each point in the grid has a value corresponding to performance for that no. of layers and nodes. We choose the point which gives best performance, i.e. minimizes test error.

References

[Back Propagation example] `https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/`