CS 337, Fall 2023

Introduction to Neural Networks

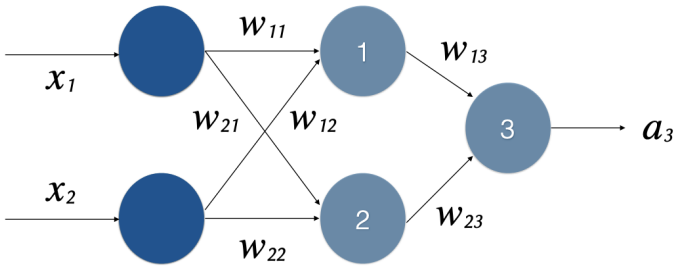
Scribes: Ameya Deshmukh, Karthikeya Satti, Rahul Deepak,
Priyanshu Yadav, TulsiRam Terli, Vinu Rakav
(***Equal contribution by all scribes***)
Edited by: Govind Saju

September 14, 2023

1 Feed-forward Neural Networks

These are quintessentially deep learning models because they involve learning the parameters to accurately match the output. The feed-forward in the name comes from the fact that in computing the output of the network, information only flows in one direction without any feedback connections. In case such feedback computations are a part of the models, they are referred to as **Recurrent Neural Networks (RNNs)**. The structure of a neural network consists of a number of *hidden* layers, each representing a vector value. Each value of the vector represents a *neuron* which acts as a vector-to-scalar function. These neuron like units of each layer work in parallel, computing their own outputs using the values of the previous layer. The ideas of having multiple layers, with each unit having an activation function are drawn from neuroscience.

1.1 A simple example



Consider the above network which computes the output a_3. For the first hidden layer [Nodes 1, 2], the

weighted input values are:

$$\begin{aligned} z_1 &= w_{11}x_1 + w_{12}x_2 + b_1 \\ z_2 &= w_{21}x_1 + w_{22}x_2 + b_2 \\ \implies \boldsymbol{z}^1 = \begin{pmatrix} z_1 & z_2 \end{pmatrix} &= \begin{pmatrix} x_1 & x_2 \end{pmatrix} \begin{pmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \end{pmatrix} + \begin{pmatrix} b_1 & b_2 \end{pmatrix} \\ \boldsymbol{z}^1 &= \boldsymbol{x}W^1 + \boldsymbol{b}^1 \end{aligned}$$

The superscript here denotes the layer number. These units apply a non-linear activation function g on these weighted sums, giving:

$$\boldsymbol{a}^1 = \begin{pmatrix} a_1 & a_2 \end{pmatrix} = \begin{pmatrix} g(z_1) & g(z_2) \end{pmatrix} = g(\boldsymbol{z}^1)$$

A similar computation happens in going from the hidden layer to the output layer [Node 3]:

$$\begin{aligned} z_3 &= w_{13}a_1 + w_{23}a_2 + b_3 \\ &= \begin{pmatrix} a_1 & a_2 \end{pmatrix} \begin{pmatrix} w_{13} \\ w_{23} \end{pmatrix} + b_3 \\ \implies \boldsymbol{z}^2 &= \boldsymbol{a}^1 W^2 + \boldsymbol{b}^2 \\ a_3 &= g(z_3) \\ \implies \boldsymbol{a}^2 &= g(\boldsymbol{z}^2) \end{aligned}$$

Combining these, we can write:

$$a_3 = g(g(\boldsymbol{x}W^1 + \boldsymbol{b}^1)W^2 + \boldsymbol{b}^2)$$

1.2 Activation functions

In general, a wide variety of differentiable functions perform perfectly well as activation functions. Some commonly used ones are:

- **Sigmoid:** $\sigma(x) = \dfrac{1}{1+e^{-x}}$
- **Hyperbolic tangent(tanh):** $\tanh(x) = \dfrac{e^{2x}-1}{e^{2x}+1}$
- **Rectified Linear Unit (ReLU):** $\text{ReLU}(x) = \max(0, x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$

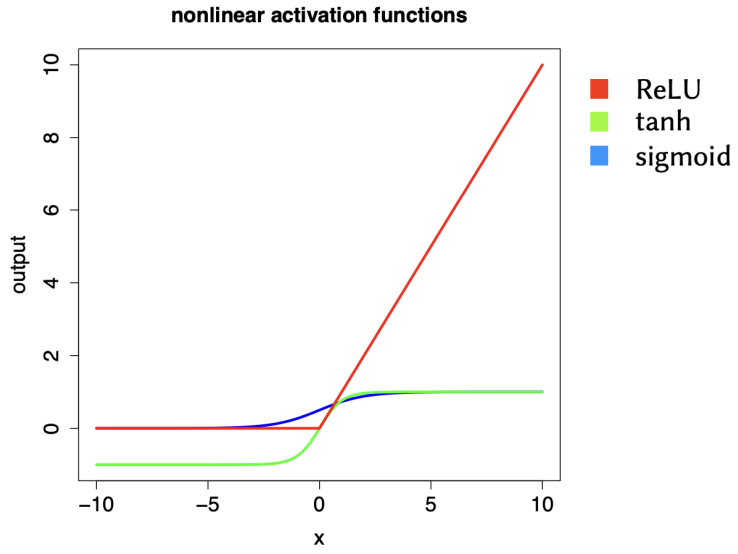
The **Rectified Linear Unit (ReLU)** activation function is easy to implement and offers better performance and generalization in deep learning as compared to the sigmoid and tanh activation functions. ReLU represents a nearly linear function and therefore preserves the properties of linear models that makes them easy to optimize with gradient descent methods. This function eliminates the vanishing gradient problem for large inputs, as observed in the other activation functions

2 Training Neural Networks

2.1 Loss function

The aim of training a NN is to minimize the loss:

$$J(\theta) = \sum_{(\boldsymbol{x_i}, y_i) \in \mathcal{D}_{\text{train}}} \mathcal{L}(\text{NN}(\boldsymbol{x_i}; \theta), y_i)$$



where θ stands for all the weights, biases $(W^l, \boldsymbol{b}^l)$ of all the layers in the network.
A popular choice for $\mathcal{L}$ in case of binary labelled data is the binary cross entropy loss:

$$J(\theta) = \sum_{(\boldsymbol{x_i}, y_i) \in \mathcal{D}_{\text{train}}} y_i \log(\text{NN}(\boldsymbol{x_i}; \theta)) + (1 - y_i) \log(1 - \text{NN}(\boldsymbol{x_i}; \theta))$$

2.2 Gradient Descent

Stochastic Gradient Descent (SGD):

Inputs: $NN(x; \theta)$, Training examples $x_1, \ldots, x_n$; Outputs, $y_1, \ldots, y_n$
Loss function: $L(NN(x_i; \theta), y_i)$
    Randomly initialize θ
    do until **stopping criterion**
        Pick a training example (x_i, y_i)
        Compute the loss $L(NN(x_i, \theta), y_i)$
        Compute the gradient of L, $\nabla_\theta L$ with respect to θ:
        Update θ: $\theta \leftarrow \theta - \eta \nabla_\theta L$, where η is Learning rate
    done
Return: θ

Mini-batch Gradient Descent (GD):

Inputs: $NN(x;\theta)$, Training examples $x_1, \ldots, x_n$; Outputs, $y_1, \ldots, y_n$
Loss function: $L(NN(x_i;\theta), y_i)$
Randomly initialize θ
do until **stopping criterion**
Randomly sample a batch of training examples $\{x_i, y_i\}_{i=1}^{b}$
(where the batch size, b, is a hyperparameter)
Compute the gradient of L over the batch, $\nabla_\theta L$ with respect to θ:
Update θ: $\theta \leftarrow \theta - \eta \nabla_\theta L$, where η is Learning rate
done
Return: θ

For this, we need to find $\frac{\partial L}{\partial w}$ for every weight w, and update it as

$$w \leftarrow w - \eta \frac{\partial L}{\partial w}$$

We can efficiently compute $\frac{\partial L}{\partial w}$ using Back-Propagation algorithm.

3 Back Propagation

Training a feed-forward neural network hinges on the problem of computing the gradient of the loss function with respect to the parameters efficiently. The back propagation algorithm applies the chain rule of calculus in a specific order to accomplish this in general for any *computational graph*. First, a simple introduction to back propagation is provided, followed by a more generalised study.

3.1 Back Propagation: An overview

We need to find $\frac{\partial L}{\partial w}$ for every weight w. We will do this by first finding $\frac{\partial L}{\partial u}$ for every node u and use following to find $\frac{\partial L}{\partial w}$:

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial u} \cdot \frac{\partial u}{\partial w}$$

To efficiently compute $\frac{\partial L}{\partial u}$ for every node u, we will use **Chain rule of differentiation** recursively for every layer. If L can be written as a function of variables v_1, ..., v_n, which in turn depend on another variable u, then

$$\frac{\partial L}{\partial u} = \sum_i \frac{\partial L}{\partial v_i} \cdot \frac{\partial v_i}{\partial u}$$

Let $\Gamma(u) = \{v_1, ..., v_n\}$ represent the children of u, then the chain rule gives

$$\frac{\partial L}{\partial u} = \sum_{v \in \Gamma(u)} \frac{\partial L}{\partial v_i} \cdot \frac{\partial v_i}{\partial u}$$

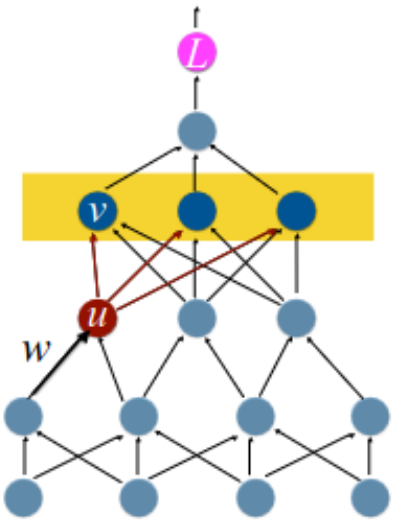


Figure 1: Image of a typical graph

In forward pass, we compute and store the activation function values for each node, to be used later in finding derivatives in backpropagation.

3.2 Back Propagation: A detailed study

Generalized chain rule

Let $g : \mathbb{R}^m \to \mathbb{R}^n$ and $f : \mathbb{R}^n \to \mathbb{R}$ be two maps. If $z = f(\boldsymbol{y}), \boldsymbol{y} = g(\boldsymbol{x})$ where $\boldsymbol{y} \in \mathbb{R}^n, \boldsymbol{x} \in \mathbb{R}^m$ then:

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}$$

$$\nabla_{\boldsymbol{x}} z = \begin{pmatrix} \vdots \\ \frac{\partial z}{\partial x_i} \\ \vdots \end{pmatrix} = \begin{pmatrix} & \vdots & \\ \frac{\partial y_1}{\partial x_i} & \cdots & \frac{\partial y_n}{\partial x_i} \\ & \vdots & \end{pmatrix} \begin{pmatrix} \frac{\partial z}{\partial y_1} \\ \vdots \\ \frac{\partial z}{\partial y_n} \end{pmatrix}$$

$$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^\top \nabla_{\boldsymbol{y}} z$$

where $\dfrac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ is the $n \times m$ Jacobian matrix of g.

Backprop for feed-forward NNs

Consider that for a single datapoint $(\boldsymbol{x}, y)$ we aim to compute the gradients of $\mathcal{L}(\text{NN}(\boldsymbol{x}; \theta), y)$. Let the NN have L layers, with the lth layer having n^l nodes, whose input is $\boldsymbol{z}^l$, and whose non-linear output is $\boldsymbol{a}^l = f^l(\boldsymbol{z}^l)$.
Based on the model seen, let $\boldsymbol{z}^l = W^l \boldsymbol{a}^{l-1} + \boldsymbol{b}^l$, and $\boldsymbol{a}^0 = \boldsymbol{x}$, $a^L = \text{NN}(\boldsymbol{x}; \theta)$.
We can now use the chain rule to compute the gradient of the loss wrt to some layer's input given the same

for the next layer.

$$\nabla_{\boldsymbol{z}^l}\mathcal{L} = \left(\frac{\partial \boldsymbol{z}^{l+1}}{\partial \boldsymbol{z}^l}\right)^\top \nabla_{\boldsymbol{z}^{l+1}}\mathcal{L}$$

Considering

$$\frac{\partial z_i^{l+1}}{\partial z_j^l} = \frac{\partial (W^{l+1}\boldsymbol{a}^l + \boldsymbol{b}^l)_i}{\partial z_j^l}$$
$$(W^{l+1}\boldsymbol{a}^l + \boldsymbol{b}^l)_i = \sum_k W_{ik}^{l+1} f^l(z_k^l) + b_i^l$$
$$\implies \frac{\partial z_i^{l+1}}{\partial z_j^l} = W_{ij}^{l+1} \cdot (f^l)'(z_j^l)$$

Hence,

$$\frac{\partial \boldsymbol{z}^{l+1}}{\partial \boldsymbol{z}^l} = W^{l+1} \cdot (f^l)'(\boldsymbol{z}^l)$$

where $(f^l)'(\boldsymbol{z}^l)$ is $\text{diag}((f^l)'(z_1^l), \ldots, (f^l)'(z_{n^l}^l))$, which finally gives us:

$$\boxed{\nabla_{\boldsymbol{z}^l}\mathcal{L} = (f^l)'(\boldsymbol{z}^l) \cdot (W^{l+1})^\top \cdot \nabla_{\boldsymbol{z}^{l+1}}\mathcal{L}}$$

Moreover, at the last layer [assumed to have a single node]:

$$\begin{aligned}\nabla_{z^L}\mathcal{L}(\text{NN}(\boldsymbol{x};\theta), y) &= \nabla_{z^L}\mathcal{L}(f^L(z^L), y) \\ &= \left.\frac{\partial \mathcal{L}(\hat{y}, y)}{\partial \hat{y}}\right|_{\hat{y}=\text{NN}(\boldsymbol{x};\theta)} \cdot (f^L)'(z^L)\end{aligned}$$

Now we can utilize these gradients to compute the gradients of the loss for the parameters. Consider any weight W_{ij}^l, it contributes to z_i^{l+1} weighted with a factor of a_j^l:

$$\implies \frac{\partial \mathcal{L}}{\partial W_{ij}^l} = \frac{\partial \mathcal{L}}{\partial z_i^{l+1}} a_j^l$$
$$\boxed{\nabla_{W^l}\mathcal{L} = \nabla_{\boldsymbol{z}^{l+1}}\mathcal{L} \cdot (\boldsymbol{a}^l)^\top}$$

Similarly for the biases, they contribute with a factor of 1 in the same manner as above:

$$\implies \frac{\partial \mathcal{L}}{\partial b_i^l} = \frac{\partial \mathcal{L}}{\partial z_i^{l+1}}$$
$$\boxed{\nabla_{\boldsymbol{b}^l}\mathcal{L} = \nabla_{\boldsymbol{z}^{l+1}}\mathcal{L}}$$

3.3 The algorithm

As seen above, the computation of the gradient can be done in a backwards manner, but it relies on both of the $\boldsymbol{a}^l$ and $\boldsymbol{z}^l$ values at each layer. These are computed in a forward pass.
Once we have the necessary gradients, the next step is gradient descent:

$$\theta \leftarrow \theta - \eta \nabla_\theta L$$

If the gradient is computed over a randomly sampled $(\boldsymbol{x}_i, y_i)$ from the training dataset, it is referred to as **Stochastic Gradient Descent (SGD)**.
Otherwise, if the gradient is computed over a randomly sampled *batch* $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^b$, it is called **Mini-batch Gradient Descent**.

Algorithm 1: Back propagation

Assume that the nodes of the network are $u_1, u_2, \ldots, u_n = L$ in a topologically sorted manner, where L is the unit computing the final loss function;
Forward pass;
for $i \leftarrow 1$ **to** n **do**
  $z(u_i) = \sum_{j:u_j \in \text{Parents}(u_i)} w_{ij} a(u_j)$;
  $a(u_i) = g(z(u_i))$
end for
Backward pass;
for $i \leftarrow n$ **to** 1 **do**
  $\partial L/\partial z(u_i) = \sum_{j:u_j \in \text{Children}(u_i)} \partial L/\partial z(u_j) * \partial z(u_j)/\partial z(u_i)$;
end for
for w_{ij} **do**
  w_{ij} feeds into node i;
  $\partial L/\partial w_{ij} = \partial L/\partial z(u_i) * \partial z(u_i)/\partial w_{ij}$;
end for

4 Regularization

In the context of deep learning, most regularization strategies are based on regularizing estimators. Regularization of an estimator works by trading increased bias for reduced variance. Some popular regularization techniques in deep learning:
Let us skip upon the pre-Deep Learning Era regularizations such as **Ridge & Lasso** regularizations.

4.1 Label Smoothing

Noise injection is one of the most powerful regularization strategies. **By adding randomness, we can reduce the variance of the models and lower the generalization error**. The question is how and where do we inject noise?

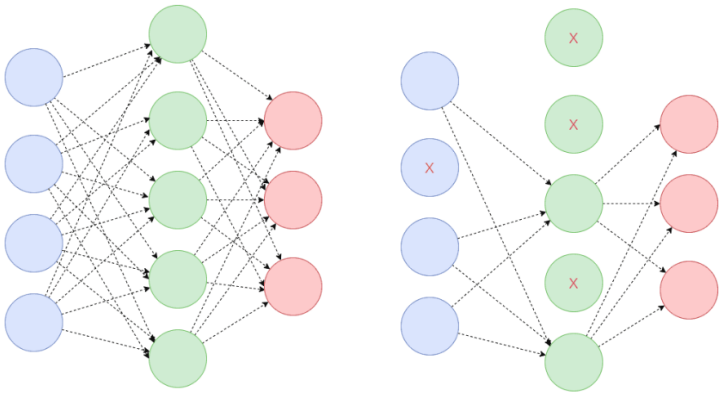
Label smoothing is a way of adding noise at the output targets, aka labels. Let's assume that we have a classification problem. In most of them, we use a form of cross-entropy loss such as $-\sum_{j=1}^{c} y_{i,j} \cdot \log(P_{i,j})$ and softmax function to output the final probabilities of labels.

If the target vector is of the form $\{0,\ 1,\ 0,\ 0\}$. Because of the way softmax is defined, it can never achieve a value of 0 or 1. Hence it continues to train on the same dataset improving upto the output probabilities to reach extremes of 0 and 1 as a result leans to the side of "overfitting" on the training data.

To address this issue, label smoothing introduces a small margin ϵ over the hard 0 and 1 in the output vectors. Specifically 0 is replaced by $\dfrac{\epsilon}{1-c}$ and 1 by 1-ϵ where c is number of classes.

4.2 Dropout

Dropout falls into noise injection techniques and can be seen as **noise injection into the hidden units of the network.** During training, some neurons of hidden layers are randomly dropped out for a forward pass with probability p
.
During test time though, all units will be present but their outputs will be scaled down by a factor of p. This is done to achieve the model similar to one we used in training.



By using dropout, the same layer will alter its connectivity and will search for alternative paths to convey the information to the next layer. As a result, each update to a layer during training is performed with a different "view" of the configured layer.
Conceptually, it approximates training a large number of neural networks with different architectures in parallel.

Dropout has the effect of making the training process noisy.This conceptualization suggests that perhaps dropout breaks up situations where network layers co-adapt to correct mistakes from prior layers, making the model more robust. It increases the sparsity of the network and in general, encourages sparse representations!

4.3 Stochastic Depth

Stochastic step goes a step further and drops entire network layers while keeping them intact during testing.In particular, it drops layers that have *Residual connections* with a probability p that is a function of **depth** of that layer.

4.4 Early Stopping

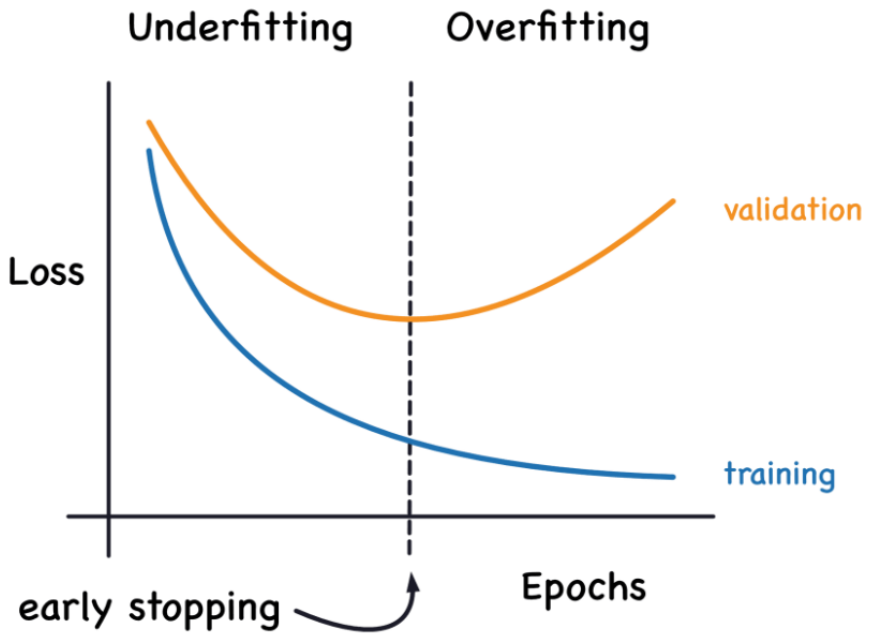
Perhaps the most commonly used regularization technique due to it's simplicity and effectiveness.It refers to the process of **stopping training when validation error is no longer decreasing.**

A different way to think of early stopping is as a very efficient hyperparameter selection algorithm, which sets the number of epochs to the absolute best. It essentially restricts the optimization procedure to a small volume of the trainable parameters space close to the initial parameters.

5 Scheduling Learning Rates

As you have guessed, it is trivial in employing same learning rate to all the features of inputs. This however need not capture all the properties of different features. We want learning rate to be responsive in each feature and time and hence several methods are employed in scheduling learning rates. Some examples are:

- Step Decay
- Exponential Decay
- Adagrad



- Adam

Adagrad and *Adam* are adaptive learning rate methods in the sense that they tune learning rates independently for different features wrt time.

6 Trivia + Practice Problems

An interesting thing pointed out at the start of the lecture: Neural networks that use linear activation functions but are still non-linear due to float precision.
Question 1: The Sum of square errors on a training data using weights after minimizing the Ridge is at least as large as the sum of squared error using weights obtained by minimizing the ir-regularized Linear Regression

Solution: True,

Ridge regression imposes a penalty on the weights and tries to minimize them, so the weights calculated using ridge regression are sure to be less than the weights calculated using the non-regularized loss. Let W_r be the weights calculated by ridge regression, and W_o be the non-regularized weights.

$$W_r \leq W_o$$

Then:

$$\|y - W_r^T x\|_2^2 \geq \|y - W_o^T x\|_2^2$$

Since $W_o = \arg\min_w \|y - W^T x\|_2^2$, by definition, the inequality above holds true.
Question 2: Let Kernel Function K be defined as $K(X, X') = \exp\left(-\frac{(X-X')^2}{2}\right)$, Let ϕ be the underlying function for K then what is the value of the expression $\|\phi - \phi'\|^2 + \|\phi + \phi'\|^2$
Solution: 4

The given kernel function $K(X, X') = \exp\left(-\frac{(X-X')^2}{2}\right)$ is a Gaussian kernel.

First, let's calculate $||\phi - \phi'||_2^2$:

$$\begin{aligned} ||\phi - \phi'||_2^2 &= \langle \phi - \phi', \phi - \phi' \rangle \\ &= \langle \phi, \phi \rangle - 2\langle \phi, \phi' \rangle + \langle \phi', \phi' \rangle \end{aligned}$$

Similarly,

$$||\phi + \phi'||_2^2 = \langle \phi, \phi \rangle + 2\langle \phi, \phi' \rangle + \langle \phi', \phi' \rangle$$

The resulting sum of both the equations is:

$$||\phi - \phi'||_2^2 + ||\phi + \phi'||_2^2 = 2\langle \phi, \phi \rangle + 2\langle \phi', \phi' \rangle$$

From the given kernel function, K(X, X) is the dot product of two vectors which is given by,

$$K(X, X) = \exp\left(-\frac{(X - X)^2}{2}\right) = \exp(0) = 1$$

Therefore,

$$2\langle \phi, \phi \rangle + 2\langle \phi', \phi' \rangle = 2(1) + 2(1) = 4$$

Question 3: In the soft-margin SVM, where each (x_i, y_i) is associated with a ξ_i and the objective contains a term $C\sum_i \xi_i$. Suppose we impose an additional constraint that for all i, $\xi_i = \xi_1$. What can we say about the minimum value of the objective function under the modified constraints, denoted as α, in comparison to the minimum under the original constraints, denoted as β?
Solution: $\alpha \geq \beta$

Question 4: Consider a data set D. Define $K : 2^D \times 2^D \to \mathbb{R}$ as $K(A, B) = |A \cap B|$. Prove/Disprove that K is a valid kernel function.

Solution: K is a valid kernel since we can explicitly define a mapping ϕ which takes every subset to a string of 0/1 with 1's at the position of every element in it.

Question 5: During construction of Decision Tree, Whenever a set S of labelled instances is split into S1 & S2. The average entropy will always decrease True/False ?

Solution: False
The average entrophy may decrease or remain constant after splitting the set S into S1 and S2. When a split is made such that all the labelled instances goes into the split S1 and no instances goes into S2, the average entropy doesn't decrease.