DEEP LEARNING: ALL QUESTIONS AND ANSWERS

Explain which: (1) activation function for the output layer, and (2) loss function, should we choose when facing a regression problem, according to the maximum likelihood principle. Motivate your answer, explaining which assumptions we are making and how we are modeling the output of the neural network from a probabilistic perspective.

What is the difference between backpropagation and SGD?

Backpropagation is a phase of the algorithm used to train a neural network. The phases are: forward pass, backpropagation and weight update.

In the forward pass we compute the output from the input. Then we compute the loss between the generated output and the true value. Backpropagation computes the gradient of the loss function for each weight by exploiting the chain rule. In order to do that backpropagation starts from the last layer and it goes backward computing the derivatives of the loss function.

Gradient descent is the method to compute a solution for a problem given its loss function. (Explain how GD works)

Stochastic GD computes the gradient of the loss on a single random example of the training set for updating the weights. SGD computes a gradient that is not accurate because it uses just a small fraction of data for computation but it's faster than traditional GD.

We can solve this problem by using minibatch GD that computes the gradient over a subset of samples rather than just one.

Why is it convenient to use more than one hidden layer in a neural network?

As stated by the universal approximation theorem a feedforward nn with a linear output layer and at least one hidden layer with any "squashing" activation function can approximate any continuous function with any desirable non-zero amount of error, given enough hidden units.

One could say that a deep nn with just one hidden layer is enough to solve any problem, but having just one layers can presents two problems: the number of hidden units can be too large which would make learning infeasible and the network is not guaranteed to generalize the function correctly.

For this reason using more than just one hidden layer is useful because the number of parameters can be reduced exponentially and a deeper network has the ability to capture the simpler features of the input.

Why do we need to use more that one layer in neural networks? In other words, what is the advantage of a multi-layer neural network over a perceptron?
(Question similar to the previous one but in this case they ask to describe the difference between a nn and a linear model)
The perceptron is a linear model that is able to separate linearly a dataset into two classes. Linear models like the perceptron are not able to generalize accurately non-linear functions because a non-

linear function is not a composition of linear functions. Also it is known that a perceptron is not able to solve the XOR problem.

A nn, instead, is able to learn a non-linear function because the activation functions on the hidden units introduce non-linearity on the network.

Explain what is dropout, how it is implemented in training and explain why does it act as a regularization

Dropout is a regularization technique where multiple smaller models are generated from the original network and evaluated on each test example. Specifically, dropout trains the ensemble (the aggregation of these smaller models representing the sub-networks) that can be formed by removing non-output units from the underlying network. For this reason dropout can be thought as a bagging tecnique but, as we will see, with some important differences.

In most neural networks we can remove a unit by multiplying its output by 0.

To train a nn with dropout we use a minibatch-based learning algorithm; each time we load an example into the minibatch, we randomly sample a different binary mask to apply to all inputs and hidden units in the network. The probability of sampling one unit can be seen as a fixed hyperparameter.

Dropout training is not exactly equal to bagging because in the latter one usually models are independent from each other, instead in dropout models usually share parameters inherited from the parent nn. This parameter sharing makes it possible to represent an exponential number of models with a tractable amount of memory.

To make a prediction a bagged ensemble must accumulate votes from all its members.

$$P_{ensemble} = \frac{1}{k}\sum_{i=1}^{k} P^{(i)}(y|x)$$

A key insight of dropout is that we can approximate $P_{ensemble}$ by evaluating $P(y|x)$ in one model, the model with all units, but with the weights of unit i multiplied by the probability of including unit i. The motivation behind it is to capture the right expected value of the output from that unit. This is called weight scaling.

Finally the advantages of dropout is that is computationally very cheap and it works well on any model that can be trained with SGD.

Dropout acts as regularization because it forces the network to learn a generalization of the target function by limiting the numbers of units and consequently the parameters.

What are the main problems in optimizing deep neural networks? Why are they a problem for optimization algorithms? How is it possible to avoid them?

The main problems are: ill conditioning the Hessian, local minima, saddle points, flat regions, cliffs, long term dependencies and inexact gradients.

The Hessian matrix H of the loss function is said to be ill-conditioned if the conditioning number, which is the ratio between the highest eigenvalue and the lowest eigenvalue of H.

A high condition number means that the loss function changes rapidly when the input is perturbed slightly so the Hessian would be very sensitive to initial inputs and round off errors during the calculation of the solution.
Since eigenvalues tell the magnitude and orientation of the curvatore of the loss, we can say that the loss has no uniform curvature. 1st order methods are not able to take the best descent direction because the loss increases rapidly with small stepsizes due to the unregular shape of the loss.

A point x is a local minima if with respect to its neighbourhood the loss function has the smallest value.
During training local minima can pose a problem because they can be associated with a high value loss: this mean that the solution we found is very far from the global optimum.
Any deep nn is guaranteed to have an extremely large number of local minima but as new research suggest for large enough nn local minima have generally low cost.

Flat regions are regions where the loss has constant value. In these regions gradient and Hessian are zero.
This is a problem in convex optimization because flat regions consist entirely of global minima and in general optimization correspond to a high-value function.

Cliffs are extremely steep regions. They may pose a problem during the computation of the gradient as the update can move the gradient far beyond the cliff (the gradient is calculated as a difference between two points and in cliffs the difference is very high).
The problem can be solved using gradient clipping, by reducing the step size when it becomes too big.
Cliffs are more common in RNNs because they involve the multiplication of several large weights.

We incur in vanishing and exploding gradient when we have long term dependencies. Consider a nn, for example a RNN, where we multiply the weight matrix W many times:

$$W = Vdiag(\lambda)V^{-1}$$

$$W^t = Vdiag(\lambda)^t V^{-1}$$

We apply spectral decomposition on W and we see that the eigenvalues are elevated to t.

- $\rho(W) < 1$

  If the spectral radius of W is lower than zero if we continue to multiply by W the gradient will eventually reach 0.
  (The spectral radius is the absolute value of the highest eigenvalue)

- $\rho(W) > 1$

  Otherwise if the spectral radius of W is greater than zero the gradient will explode.

This problem can be solved algorithmically by using Hessian-free optimization and gradient clipping or in the context of RNNs by architectural means: LSTMs and Reservoir Computing.

Inexact gradient. Traditional optimization algorithms assume exact knowledge of the gradient while we usually only a biased or noisy estimation of it (Minibatch GD).
In other cases the actual objective function and consequently the gradient are actually intractable. This is usually true for RBMs.
This problem can't be solved because it is a problem that derives from the need to balance a good estimation to compute the result faster. It really depends on our goals.

Explain the main characteristics and the differences among: gradient descent, minibatch gradient descent , AdaGrad, RMSprop and Adam.

(explain GD and minibatch GD)

AdaGrad is an optimization algorithm that scales the learning rate inversely proportional to the square root of the sum of all historical values of the gradient. The algorithm is as follows:

$$compute\ gradient\ g$$
$$accumulate\ squared\ gradient\ r \leftarrow r + g \odot g$$
$$update\ rule\ \Delta\theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{r}} \odot g$$

AdaGrad shrinks the learning rate accordine to the entire history of the gradient and may reduce the learning rate too much before getting to the solution.
RMSprop discards the history from the extreme past by using an exponential decaying average.

$$compute\ gradient\ g$$
$$r \leftarrow \rho r + (1-\rho) g \odot g$$

Adam is a variant of RMSprop and momentum. In Adam momentum is incorporated directly as an estimate of the first order moment of the gradient. Secondly it includes bias correlation to the estimate of both first order and second order moments.

Explain why it is not a good idea to initialise all the weights of a Neural Network to zero. Consider both the cases in which the ReLU activation function is used, and the general case. Detail your answer and provide examples.
Training a nn depends on the initialization of parameters. In some cases a bad initialization can make the network unstable.

One of these cases is when we have symmetry, which happens when we initialize all the weights with the same value (in this case 0). When we perform backpropagation the update will be the same for each parameter and the weights will remain equal to each other. For this reason the nn will collapse to only one neuron, obviously not a good outcome.

If we have ReLU as an activation function and all the weights are set to 0 we incur in another problem bacause ReLU presents a discontinuos point in 0. The calculated gradient will be discontinuous as well.

Give the definition of sequential transductions, explaining the concept of memory, causality and recursive state representation. For each different type of sequential transductions we discussed describe a network architecture able to implement it.
A sequential transduction is a function that given in input a sequence gives in output possibly another sequence based on the relationship established by the input.
Let X and O be input and output label spaces. X^* is the set of sequences with labels in X. A sequential transduction is $X^* \rightarrow O^*$.

A transduction has finite memory if for each element of the sequential data the transduction of that element only depens on a finite number of previous data.

A transduction is causal if the output at time t does not depend on future outputs.

We define the recursive state representation as:

$$h^{(t)} = f\left(h^{(t-1)}, x^{(t)}\right)$$

$$o^{(t)} = g\left(h^{(t)}, x^{(t)}\right)$$

Here we have a hidden unit which correspond to the internal state representation of the previous inputs. The hidden unit at time t depends only on the hidden state at time t-1 and the input at time t. The recursive state representation is only possible if the transduction is causal.

- Sequence transduction (n to m transduction): Transformer, RNN
- Sequence classification (n to 1 transduction): CNN, RNN or bidirectional RNN
- Sequence generator (1 to n transduction): a CNN in pair with a RNN
- IO-transduction (n to n transduction): Encoder/Decoder

In the context of sequential transductions, give the definition of causality and describe how this concept is used in RNNs. Are all RNNs architectures causal?
Causality refers to the notion that the output at a given time step should only depend on the previous time steps and not the future ones. In other words the order of the input sequence is preserved and the information flows only from past to future.

RNNs are a class of nn commonly used for modeling sequential data data. RNNs implement causality through the use of recurrent connections, which allow information to be passed from one time step to the next.

The key mechanism that enables causality in RNNs is the hidden state, which serves as an internal representation of the past inputs. The hidden state is updated at each time step based on the current input and previous hidden state. This allows the network to capture and remember information from the past, which can influence the predictions at future time steps.

$$h^{(t)} = f\left(x^{(t)}, h^{(t-1)}\right)$$

This implementation ensures causality since the prediction at time step t depends only on the hidden state at time step t-1 and the input at t.

Not all RNNs are strictly causal. Certain variations like bidirectional RNNs and RNNs with attention mechanisms may introduce connections that allow the model to access future information.

Introduce the GRU formula and explain it.
A GRU is a simpler version of the LSTM where one gate controls the forgetting factor and the ability t update the stored value.

$$h^{(t)} = z^{(t)} \odot h^{(t-1)} + \left(1 - z^{(t)}\right) \odot \sigma(Ux^{(t)} + W(r^{(t)} \odot h^{(t-1)}))$$

For z = 1 we store the information from the past (memory) because the internal state depends only on the previous time step t-1.

$$h^{(t)} = h^{(t-1)}$$

For z = 0 and r = 0 the internal state depends only on the input, so we are in the case when we forget and we update the memory cell.

$$h^{(t)} = \sigma(Ux^{(t)})$$

For z = 0 and r = 1 it behaves just like a vanilla RNN.

$$h^{(t)} = \sigma(Ux^{(t)} + Wh^{(t-1)})$$

Explain what a Reservoir Computing is. What are the main features and properties that such model owns?

RC is an architecture that addresses the problems of exploding and vanishing gradient that occur usually in Recurrent Neural Networks.

The main idea of RC is to generate a network where only the output connections are known and the input-to-hidden and hidden-to-hidden connections are not. The latter are initialized randomly in the network and because there aren't ways to implement memory, loops are allowed (similar to Flip-Flop architecture).

In order to produce a rich set of dynamics the network should be big, sparse and preserve the echo state property. The echo state property is the ability to forget information from the far past (or the effect of x(t) and h(t) on the future state should vanish gradually as time passes).
Mathematically this property is preserved if $\rho(W) < 1$ (matrix W is contractive).

To evaluate a RC network we use memory capacity, which tells us if the internal state of the network can reproduce input from the far past i.e. does it memorizes past information?

$$\sum_{r=0}^{\infty} r^2 \left(o_k^{(t)}, x^{(t-k)}\right)$$

Another feature of RC is intrinsic plasticity, which refers to the ability of the RC to self adapt and change its internal dynamics in response to input patterns without the need of external feedback.
How? Nodes approximate an exponential distribution, which maximizes the entropy of a non-negative random variable thus enabling nodes to transmit maximum information.

A RC can be implemented as an Echo State Network plus a leaky integrator.
Many possible topologies for a RC have been tried but only simpler ones improved performances.
Finally, a deep RC can be made by stacking together many simpler RC networks.

What is the difference between a probabilistic model represented by a directed graph and one represented by an undirected graph?

A naive approach is the table based approach where we explicitly model every possible kind of interaction between every possible subset of variables. This is computationally intractable because the computionally is exponentially large.
Probabilistic models are models used to represent a probability distribution in an efficient by just representing the interactions between variables. We can perform probabilistic inference with any variable of the distribution by the joint probability distribution.
Probabilistic models use graphs to represent probability distributions where every node is a variable and each edge represents an interaction between two variables.

Directed models represent interactions with directed edges. For example is there is an edge between a variable A and a variable B that means that the distribution of B depends on A.
The probability distribution of the network is derived by multiplying the probability of each node given its parent.

$$P(x_1, \dots, x_n) = \prod_{i=1}^{n} P(x_i | parent(x_i))$$

Undirected models, instead, are used when we don't know the direction of the dependencies. In this case we define a clique potential for each clique (variables that influence each other and form a connected component) a clique potential greater or equal to 0 and we compute the unnormalized probability distribution.
Then we normalize it by dividing it by Z, the partition function that is calculated as the integral over the unnormalized probability distribution.

$$\tilde{P}(x) = \prod_{C \in G} \phi(x)$$

$$P(x) = \frac{1}{Z}\tilde{P}(x) \; where \; Z = \int \tilde{P}(x) dX$$

A special case of probabilistic models that use undirected graphs are Energy-based models which use a energy function to compute P.

$$P(x) = e^{-E[x]}$$

Calculating Z as an integral is not feasible because we would have to consider all the variables x and that is not reasonable. For this reason Z is usually estimated by sampling; an example of that is Gibbs Sampling used in RBMs.

Explain in detail what is the role of Monte Carlo Methods in training stochastic neural networks. Give and example of a neural network where Monte Carlo Chain is used.
Monte Carlo Methods are methods to estimate the probability of a variable that comes from a probability distribution. They are useful to Stochastic Neural Networks because in a nn we never get an exact solution but we always approximate it.
MCMs are preferred to Las Vegas methods because LA methods find the correct solution, which is nearly impossible to get in a nn.

For probabilistic models we need to maximize the log-likelihood of p(x) but the gradient of it

$$\nabla \log \mathrm{p}(x) = \nabla log \tilde{p}(x) - \nabla log Z$$

Involves the term Z which is equal to $Z = \int \tilde{p}(x) dx$.

MCMs use sampling because sometimes the computation of a sum or integral is intractable.

$$s = E_p[f(x)] = \int p(x) f(x) dx$$

Given x_1, ..., x_n samples from a probability distribution p we approximate the solution by calculating the sample mean

$$\hat{s} = \frac{1}{n}\sum_{i=1}^{n} E[f(x^{(i)})]$$

(Explain the properties: unbiase, variance tends to 0, LLN)

In some cases we can't directly sample from p but we can use a trick where we make q a new distribution where we can sample from.

$$p(x)f(x) = q(x)\frac{p(x)f(x)}{q(x)} \; and \; q^* = \frac{p(x)|f(x)|}{Z}$$

$$\hat{s} = \sum_i \frac{p(x)f(x)}{q(x)}$$

A neural network that uses a Monte Carlo Chain is the RBM that uses Gibbs sampling.

(I'm not very sure about this answer)

Define a sparse autoencoder. Explain the differences with respect to an undercomplete autoencoder.

A sparse autoencoder is a overcomplete autoencoder where we add a penalty term in the reconstruction loss. A overcomplete autoencoder is a autoencoder in which the dimension of the hidden layer is greater or equal the dimension of the input.
Instead of just copying the input to the output the regularizing term enforces the network to learn the most important features of the input as a byproduct. We use L1 regularization to enforce sparsity.

$$P(h_i) = \frac{\lambda}{2}e^{-\lambda|h_i|}$$

$$-logP(h) = -\log\sum_i P(h_i) = \sum_i \left(\lambda|h_i| - \log\frac{\lambda}{2}\right) = \Omega(h) + constant$$

We can see that sparsity as a result from the effect of $P_{model}(h)$ to approximate maximum likelihood. From this point of view the sparsity term is not a regularization term at all but is a consequence of the model distribution over its latent variable.
A sparse autoencoder is a way to approximately train a generative model.

Undercomplete autoencoders learn an encoding of the input by limiting their architecture. These architectures are composed by one (or more) fully connected layers of decreasing size, up to the encoding layer which must be smaller than the input, so that the network is forced to select some relevant features.
Then, a mirrored-shaped dense feedforward network produces the reconstruction of the input based on the encoding.

These networks have traditionally been used for feature selection and dimensionality reduction, and when the decoder is linear with MSE loss they are actually equivalent to PCA techniques. Indeed, they learn to project the input data over the dimension which has greatest variance, and therefore carries the most information of the input data.

Define a denoising autoencoder.
A denoising autoencoder is a autoencoder that receives corrupt input data and is trained to predict the original uncorrupted data inputs.
We introduce a corruption process $C(\tilde{x}|x)$ which is a conditional distribution over corrupted samples.
The autoencoder learns a reconstruction distribution $P_{reconstruct}(x|\tilde{x})$ estimated from the training pairs $(\tilde{x}|x)$.

Define a contractive autoencoder.
A contractive autoencoder is designed to learn a compressed representation of data while being robust to small perturbations on the input.
It achieves this by incorporating a regularization term into its loss which encourages the model to learn features that are insensitive to small changes in the input space.
To enforce the contractive property we compute the penalty as the Frobenius norm of the derivative of the encoder's hidden layer activations with respect to the input data.

$$\Omega(h) = \lambda \left| \frac{\partial f(x)}{\partial x} \right|_F^2$$

The issues for this model are that the computation of the Jacobian for a nn is very intensive and the scale of the decoder should be equal to the scale of the encoder, this imposes the weight matrix of the decoder to be the transpose of the encoder.

What is a differentiable generator network? Why is it useful? Give an example of DGN.
A DGN is a neural network capable of generating new samples from a latent distribution. To do that we train the model to learn the mapping between a latent variable z and the distribution of the data x.

DGNs assume that the information carried by each sample can be represented in a simpler space compared to the data space. If the distribution of data is a high-dimensional space, the information of the image can be nevertheless expressed in a simpler manifold.

There are two ways to utilize the latent space to generate new data:
1. Transform a latent variable z into a new sample x
2. Transform z into a distribution over x (generate a function to approximate P(x) and then sample from it)

The key characteristic of DGNs is that they are designed to be differentiable end-to-end. This means that the whole network can be trained using gradient methods. This property enables the use of backpropagation efficiently to update parameters.

DGNs are useful because they don't just reconstruct the input data but because they allow us to generate new samples which are very similar to our original data.

An example of DGN is the Variational Autoencoder which produces new samples via the direct method and uses a deconder-encoder architecture.

Define a variational autoencoder. Explain why it is used and how it can be trained.
A VAE is a differentiable generator network that has a similar structure to an autoencoder which is able to generate new data from the latent space.

The architecture of a VAE is similar to a autoencoder with a decoder-encoder network but a VAE can generate new samples.
In order to do that the training data x is encoded as a distribution over the latent space. Then a point from the latent space is sampled from the distribution and it is decoded into a generated output. Finally the reconstruction error is computed and it is backpropagated through the network.

$$x \xrightarrow{q(z|x)} z \xrightarrow{p(x|z)} \hat{x}$$

The encoded distribution is chosen to be normal so that the encoder can be trained to return the mean and the covariance.

The newly generated data has to be regularized that means that data points should follow two main properties: continuity, two close points in the latent space should not give two completely different outputs and completeness, for a chosen distribution any point sampled from the latent space should give meaningful content after being decoded.

To calculate the posterior probability p(z|x) we have use the Bayes Theorem

$$p(z|x) = \frac{p(x,z)}{p(x)}$$

but in this case the computation of p(x) is intractable. To approximate L(x), the log-likelihood of p(x) we use ELBO (evidence of lower bound) which says that L(x) is lower bounded by:

$$L(x) \leq E[\log(p|x)] - D_{KL}(q(z|x)||p(h))$$

The first term is the log-likelihood of the observed data under the variational distribution. This term encourages the model to reconstruct the observed data as close as possible to the input data.
The second one is the KL divergence that encourages the variational distribution q(z|x) to follow the rugularization factors. In order to that p(h) is chosen as a standard normal distribution.
(ELBO is derived using the concept of variational inference which involves introducing a variational distribution q(z|x) that approximates the true but often intractable posterior distribution over the latent variable, given the observed data p(z|x))

As a last consideration we said that a differentiable generator network has to be differentiable end-to-end but the sampling layer is a probabilistic layer so we cannot do backpropagation because of it. To allow the computation of the gradient we use the reparametrization trick, in which we fix the probabilistic input into a standard normal and we separate it by the mean and the variance. We can compute the gradient of the mean and variance and apply backpropagation.

$$z = \mu + \sigma \odot \epsilon$$

Define a Generative Adversarial Network. Explain why is it useful and how it can be trained?
A GAN is a differentiable generator network that estimates the latent space distribution indirectly and can sample from that probability to create synthetic data. Unlike RBMs and VAEs a GAN exploit

implicit density estimation by estimating the underlying distribution indirectly via a game theory based approach in which two players play a min-max zero-sum game.

Those two are the generator and the discriminator. The generator takes random noise and generates an output that is as close as possible to the real sample. Then the generated ouput is fed to the discriminator which tries to determine which output is real or fake. This is a game theory approach because the generator and the discriminator go back and forth to deceive and expose each other.

The equilibrium is reached when the discriminator always outputs a ½ probability (i.e. it can't distinguish between true and false inputs).

(For the loss function look at the answer below)

Introduce the objective function used in GAN and explain it.
For the Decoder the objective function is equal to:

$$\min\left(E[logD(x)] + E[1 - \log\left(D(G(x))\right)]\right)$$

The first term is the function that tells us the probability of the discriminator of spotting real inputs and instead the second term is the probability of the discriminator to discriminate the fake outputs created by the generator.
The goal for the discriminator is to maximize the probability of spotting real input data and minimizing the probability of evaluating generated outputs as real.

For the generator the function is just:

$$\min\left(E[1 - \log\left(D(G)\right)]\right)$$

Its job is to maximize the probability that the discriminator will mark its generated outputs as true.
Empirically the objective function of G has a flat behavior and produces a small gradient in the region where data is poorly generated (easily spotted).
This is a problem because makes learning very slow so instead of using a minimization function we try to maximize it; this will invert the function and will make the gradient on the region much more higher.

$$-\max\left(E[1 - \log\left(D(G)\right)]\right)$$

Overall the goal of GAN is to find a Nash Equilibrium between the generator and the discriminator where G produces samples indistinguishable from real data and D can't discriminate the outputs.

However, switching between two objective functions makes training difficult for GANs. For this reasons GANs have been replaced by other models for generating data, like for example Diffusion Models.