WHAT'S THE DIFFERENCE BETWEEN BACKPROPAGATION AND SGD?

Backpropagation is a phase of the algorithm used to train a NN. The training for a NN is divided into 3 phases: forward phase, backpropagation phase and optimization (i.e. updating weights).
This last phase is performed by optimization algorithm such as SGD. Hence, backpropagation is a phase of the training process while SGD is an optimization algorithm used for the update of the weights.
After the forward phase, in which the input flows through all the hidden layers to the output layer, the network computes the output of each unit to obtain the loss. The goal of the backpropagation is to compute the gradient of the loss function with respect to the parameters in an efficient way using the chain rule. In order to do this, the backpropagation algorithm starts from the last layer, and it goes backward computing the derivatives of the loss functions that compose the gradient. Once the gradient has been computed, an optimization algorithm can use it to minimize the loss function modifying the parameters learned during the forward phase.
The SGD minimizes the loss function using only one random example (of the training set) to approximate the gradient, making the update process quicker (e.g. having a training set of 1 million examples it can perform 1 million updates for epoch). The disadvantage of the SGD is that the gradient computed in this way will not approximate well the real one.

EXPLAIN WHAT'S DROPOUT, HOW IT'S IMPLEMENTED IN BOTH TRAINING AND EVALUATION, AND WHY DOES IT ACT AS A REGULARIZATION

Dropout can be thought as a regularization technique; indeed, it can be seen as a way to perform an ensemble method on deep NNs. Dropout is performed by using binary masks that are able to redefine the neural network, silencing some units (the probability of silencing is an hyperparameters).
Dropout is an efficient way of performing Bagging, i.e. the technique for which k different dataset are created from the original one using sampling with replacement (approximatively 1/3 of the examples will be repeated if sampling n elements). These datasets are then used to train k different models that are going to vote each output. The majority of vote is used to decide the final output for an instance. In NNs, bagging can be carried out a s Dropout inactivating some of the units of the network of each model before the training phase. Usually hidden units are removed/silenced but also output units can be removed too, and in this case we talk about label smoothing.
If we want to use the Dropout method, we can adopt a mini-batch learning algorithm (apply a binary mask to all input and hidden layers, and with a weights update consisting in multiplying the weights going out from a unit by the probability of including that unit).
This process acts as a regularization technique because it's able to reduce overfitting by considering simpler models, but more of them, in a way that hopefully the mistakes of one model will not be repeated by another model.

WHAT ARE THE MAIN PROBLEMS IN OPTIMASING DEEP NNs? WHY ARE THEY A PROBLEM FOR OPTIMIZATION ALGORITHMS? HOW IS IT POSSIBLE TO AVOID SUCH PROBLEMS?

Optimising a deep NN is difficult due to various problems. The first and the main one is non-convex functions: the cost function of a deep NN is quite always a non-convex function; this implies the presence of local minima and/or saddle points (it's local minimum if all eigenvalues of hessian matrix are zero, while it's saddle points if eigenvalues of hessian matrix are a mixture of positive and negative values) which are critical points (i.e. gradient is null) but they are not global minima. Thus, first order methods might get stuck in these points. This issue is usually solved by searching for points with a low cost; indeed, it's possible that in the optimization process of a NN the gradient norm does not turn into zero.

In n-dimensional spaces, the presence of flat regions might be an issue, but in this case, SGD is able to escape these areas. Another problem is represented by cliffs, i.e. regions where the gradient is huge, and the iteration might be overshoot by this huge value. In this case, a possible solution could be Clipping by gradient.
Another issue is the vanishing gradient, which is due to parameters sharing and might be solved using both architectural (e.g. GRUs) and algorithmic (e.g. Hessian Free Optimization) solutions.
Also bad initialization may be a problem, but it can be managed using random initialization (gaussian or uniform) which allows to break the symmetry.
Moreover, a serious problem is represented by ill-conditioning, and information of second order is needed in order to not get stuck in curvatures points.
Finally, inexact gradient is a problem. Indeed, in deep learning we have access to noisy/biased estimates of the true gradient, so using mini-batch SGD can help, maybe also with the implementation of the momentum.

EXPLAIN THE MAIN CHARACTERISTICS AND DIFFERENCES AMONG: GD, MINI-BATCH SGD, ADAGRAD, RMS PROP AND ADAM:

GD is the algorithm to minimize the objective function of the problem. GD computes the gradient with reference to the entire training set and then updates the weights, so it does an update per epoch but it computes the correct gradient.
Mini-batch SGD uses a small number (batch) of training samples to compute the gradient after have trained the network on that subset, and the it updates the weights. In this case in one epoch the weights are updated several times, depending on the size of the batch.
Adagrad, RMSprop and Adam are optimization algorithms as SGD, but they exploit the history and/or momentum of the gradient of the loss that is computed to scale the learning rate, so the overall optimization is faster. Adagrad add to the present gradient the squared value of the previous one, then the learning rate is scaled by 1/r^0.5 (the convergence is faster than GD and SGD). In RMSprop, instead, the old history of the gradient is discarder in favour to the recent one which leads to a more precise learning process. So, to r (the square of the past value of the gradient) we multiply p to save the recent values of the gradient. Then, the learning rate is scaled as in the Adagrad algorithm.
In the end, Adam exploits momentum and learning rate scaling, fixing the learning using also second order method for a faster computation.

EXPLAIN WHY IT IS NOT A GOOD IDEA TO INITIALIZE ALL WEIGHTS OF A NN TO ZERO:

In this case we end up in the symmetry case that has to be avoided otherwise when we update the weights of the various neurons in a hidden layer, we can have that the update is the same for each neuron. Hence, in this situation the neurons can be removed except for one. Indeed, they are computing the same function, so the layer will be equivalent to a layer with only one neuron. This situation is called symmetry and it's due to a bad initialization and it has to be solved.

EXPLAIN WHAT A RESERVOIR COMPUTING NETWORK IS. WHAT ARE THE MAIN FEATURES AND PROPERTIES THAT SUCH A MODEL OWNS?

A reservoir computing is an architectural approach to try to solve the vanishing gradient problem. The idea is to learn only the parameters between the last hidden layer and the output layer, avoiding learning the parameters of the input to hidden and hidden to hidden layers.

The reservoir computing consists of building a random RNN which can learn many dynamics or histories of the input sequence and then applying a linear predictor for the output. The random RNN is called reservoir because its target is to learn a rich set of dynamics and in order to do that it has to be big, sparse and it has to respect the echo state property that means that the effect of input and hidden state at time t (the current state) on a future state t+T should vanish gradually as time passes.
There are many architecture that can be used for the reservoir, but they are all pretty equivalent to the random RNN.
In order to evaluate the performance achieved by reservoir method, there are two possible ways: the richness of dynamics, that is the entropy of output's distribution, and the memory capacity.

WHAT IS THE DIFFERENCE BETWEEN PROBABILISTIC MODELS REPRESENTED BYA DIRECTED GRAPH AND BY AN UNDIRECTED GRAPH? EXPLAIN THE PROS AND CONS OF EACH REPRESENTATION:

The probabilistic models are models used to represent probability distribution in an efficient way representing only the interaction between variables of the distribution. Having a representation of a joint probability distribution is important because it allows us to perform probabilistic inference with any variables of the distribution.
Probabilistic models use graphs to represent probability distribution where each node is a variable, and a link represents interaction between two variables. There are directed models which use directed graphs and undirected models which use undirected graphs.
The directed model represents the interaction between variables with directed arrows, so for example if there's an arrow between A and B (pointing on B) it means that the distribution of B depends on A. With directed models we can compute the probability distribution factorizing the probability of each variable of the graph given its parents variables. Directed models are used when we know the direction of the dependency between the variables, while id we do not know this or if the dependency goes in both directions we can use undirected models, which use undirected graphs with no directed arrows between nodes. With undirected models we can compute the unnormalized probability distribution factorizing the potential of all the clique in the graph (subset of connected nodes) and using that to compute the partition function Z that is the integral of the unnormalized probability distribution. Finally, the normalized probability distribution is computed multiplying 1/Z by the unnormalized probability distribution. Another kind of undirected models are energy models which use the energy function to compute the unnormalized probability. This can be and advantage because we can select an energy function in order to make the learning more efficient.
Pros and cons: with directed models it is easier to draw variables from the probability distribution represented (we can use Ancestral Sampling) but drawing given other variables may be more difficult. With undirected models the process of drawing variables from the represented probability distribution is slower and more difficult because we have to represent the probability distribution with some Monte Carlo methods (such as Gibb Sampling).

DEFINE A CONTRACTIVE AUTOENCODER. EXPLAIN THE DIFFERENCES WITH RESPECT TO UNDERCOMPLETE AUTOENCODER:

A Contractive autoencoder is a kind of overcomplete autoencoder together with sparse and denoising autoencoder. Overcomplete autoencoder learn an encoding of the input which can be greater than the input itself, but it has some constraints. In particular, this kind of autoencoder impose a Frobenius norm-based constraint on the derivative of the encoding with reference to the input. This means that the network will privilege that do not vary much when the input varies and are thus smoother.
Some problems with this architecture are the heavy computation of the Jacobian, and the fact that the

encoder and decoder matrices must be equally scaled.
Unlike overcomplete autoencoders, undercomplete ones learn an encoding of the input which is limited by architectural means. 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 dimensionally reduction, and when the decoder is linear with MSE loss they are 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 GAN. EXPLAIN WHY IT IS USEFUL AND HOW IT CAN BE TRAINED:

A GAN is a generative model, i.e. it tries to estimate the true probability behind the input data, so that it can sample from the probability to extract new, synthetic samples which are as close as possible to the true one. Unlike RBMs and VAEs, GANs exploit implicit density estimation, i.e. they do not estimate directly the distribution under the data, but sample from it indirectly.
In particular, GANs exploit a game-theoretically based approach, in which they solve a problem equivalent to a minmax zero-sum game between two players: a generative network (generator) and a discriminative one (discriminator). The first one generates fake samples trying to fool the discriminator, while the second one tries to distinguish between real and fake samples. The architecture of a GAN is so composed: it's initially extracted a sample from random noise, and then a deep generator network learns to transform it in a sample which is as similar as possible to true samples. Then, the generated samples are fed into the discriminator together with true samples, and the discriminator compares them trying to understand which is which.
The equilibrium of the network is reached when the discriminator always outputs ½ probability (meaning that it cannot distinguish between real and fake inputs). The optimization process alternatively optimizes the generator and discriminator loss. The latter one is: $\log(D(x))+\log(1-D(G(x)))$ and it has to be maximized; instead the generator loss is: $\log(1-D(G(x)))$ and it has to be minimized.
In practice, the generator optimization problem is replaced with $-\log(D(G(x)))$ due to performance reason. Indeed, this loss leads to faster training, however, the alternation between the optimization of the two functions makes this kind of networks hard to train.