**Explain what is dropout, how it is implemented in both training and evaluation, and why it does act as a regularization.

7.12 Dropout

What is dropout?

Dropout provides a computationally inexpensive but powerful method of regularizing a broad family of models. To a first approximation, dropout can be thought of as a method of making bagging practical for ensembles of very many large neural networks.

Bagging involves training multiple models and evaluating multiple models on each test example. This seems impractical when each model is a large neural network, since training and evaluating such networks is costly in terms of runtime and memory. It is common to use ensembles of five to ten neural networks—Szegedy et al. (2014a) used six to win the ILSVRC—but more than this rapidly becomes unwieldy. Dropout provides an inexpensive approximation to training and evaluating a bagged ensemble of exponentially many neural networks.

Specifically, dropout trains the ensemble consisting of all subnetworks that can be formed by removing nonoutput units from an underlying base network. In most modern neural networks, based on a series of affine transformations and nonlinearities, we can effectively remove a unit from a network by multiplying its output value by zero. Here, we present the dropout algorithm in terms of multiplication by zero for simplicity, but it can be trivially modified to work with other operations that remove a unit from the network.

Recall that to learn with bagging, we define k different models, construct k different datasets by sampling from the training set with replacement, and then train model i on dataset i.

Why does it act as a regularization?

Dropout aims to approximate this process, but with an exponentially large number of neural networks. Specifically, to train with dropout, we use a minibatch-based learning algorithm that makes small steps, such as stochastic gradient descent. Each time we load an example into a minibatch, we randomly sample a different binary mask to apply to all the input and hidden units in the network. The mask for each unit is sampled independently from all the others. The probability of sampling a mask value of one (causing a unit to be included) is a hyperparameter fixed before training begins. It is not a function of the current value of the model parameters or the input example. Typically, an input unit is included with probability 0.8, and a hidden unit is included with probability 0.5. We then run forward propagation, back-propagation, and the learning update as usual. Figure 7.7 illustrates how to run forward propagation with dropout.

More formally, suppose that a mask vector μ specifies which units to include, and J(θ, μ) defines the cost of the model defined by parameters θ and mask μ.
Then dropout training consists of minimizing E μ J (θ, μ). The expectation contains exponentially many terms, but we can obtain an unbiased estimate of its gradient by sampling values of μ.

How it is implemented in training?

Dropout training is not quite the same as bagging training. In the case of bagging, the models are all independent. In the case of dropout, the models share parameters, with each model inheriting a different subset of parameters from the parent neural network. This parameter sharing makes it possible to represent an exponential number of models with a tractable amount of memory.
In the case of bagging, each model is trained to convergence on its respective training set. In the case of dropout, typically most models are not explicitly trained at all—usually, the model is large enough that it would be infeasible to sample all possible subnetworks within the lifetime of the universe. Instead, a tiny fraction of the possible subnetworks are each trained for a single step, and the parameter sharing causes the remaining subnetworks to arrive at good settings of the parameters. These are the only differences. Beyond these, dropout follows the bagging algorithm. For example, the training set encountered by each subnetwork is indeed a subset of the original training set sampled with replacement.

How it is implemented in evaluation?

To make a prediction, a bagged ensemble must accumulate votes from all its members. We refer to this process as inference in this context. So far, our description of bagging and dropout has not required that the model be explicitly probabilistic. Now, we assume that the model's role is to output a probability distribution. In the case of bagging, each model i produces a probability distribution p (i)(y | x).
distributions, 1/k sum i=1until k, p (i) (y | x). (7.52)
In the case of dropout, each submodel defined by mask vector μ defines a probability distribution p(y | x, μ). The arithmetic mean over all masks is given by sum:p(μ)p(y | x, μ), (7.53)
where p(μ) is the probability distribution that was used to sample μ at training time. Because this sum includes an exponential number of terms, it is intractable to evaluate except when the structure of the model permits some form of simplification. So far, deep neural nets are not known to permit any tractable simplification. Instead, we can approximate the inference with sampling, by averaging together the output from many masks. Even 10–20 masks are often sufficient to obtain good performance.

An even better approach, however, allows us to obtain a good approximation to the predictions of the entire ensemble, at the cost of only one forward propagation. To do so, we change to using the geometric mean rather than the arithmetic mean of the ensemble members' predicted distributions.

present arguments and empirical evidence that the geometric mean performs comparably to the arithmetic mean in this context. The geometric mean of multiple probability distributions is not guaranteed to be a probability distribution. To guarantee that the result is a probability distribution, we impose the requirement that none of the submodels assigns probability 0 to any event, and we renormalize the resulting distribution. The unnormalized probability distribution defined directly by the geometric mean is given by

where d is the number of units that may be dropped. Here we use a uniform distribution over μ to simplify the presentation, but nonuniform distributions are also possible. To make predictions we must renormalize the ensemble.

A key insight (Hinton et al., 2012c) involved in 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 going out of unit i multiplied by the probability of including unit i. The motivation for this modification is to capture the right expected value of the output from that unit. We call this approach the weight scaling inference rule. There is not yet any theoretical argument for the accuracy of this approximate inference rule in deep nonlinear networks, but empirically it performs very well.

Because we usually use an inclusion probability of 1/2, the weight scaling rule usually amounts to dividing the weights by 2 at the end of training, and then using the model as usual. Another way to achieve the same result is to multiply the states of the units by 2 during training. Either way, the goal is to make sure that the expected total input to a unit at test time is roughly the same as the expected total input to that unit at train time, even though half the units at train time are missing on average.

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

8.2.1 Ill-Conditioning:
Hessian matrix is the second order derivative of the function which shows us the curvature of the loss function. When we have ill conditioning of the hessian matrix, it means that the Hessian is very sensitive to changes in input data and the round off errors made during the solution procedure. It can make the SGD to be stuck meaning it would make increase the loss function a lot even when we are taking very small steps.

To find out if ill-conditioning is harmful to a neural network training task, we can monitor the squared gradient norm (g.T g) and the (g.T Hg) term. In many cases, the gradient norm does not shrink significantly throughout learning, but the g.T Hg term grows by more than 10 times w.r.t gradient norm. The result is that learning becomes very slow despite the presence of a strong gradient because the learning rate must be shrunk to compensate for even stronger curvature.

8.2.2 Local Minima:
Local minima can be problematic if they have high cost in comparison to the global minimum. One can construct small neural networks, even without hidden units, that have local minima with higher cost than the global minima.

If local minima with high cost are common, this could pose a serious problem for gradient-based optimization algorithms. Whether networks of practical interest have many local minima of high cost and whether optimization algorithms encounter them remain open questions. For many years, most practitioners believed that local minima were a common problem plaguing neural network optimization. Today, that does not appear to be the case.
The problem remains an active area of research, but experts now suspect that, for sufficiently large neural networks, most local minima have a low cost function value, and that it is not important to find a true global minimum rather than to find a point in parameter space that has low but not minimal cost.

8.2.3 Plateaus, Saddle Points and Other Flat Regions:
another kind of point with zero gradient
Some points around a saddle point have greater cost than the saddle point, while others have a lower cost. At a saddle point, the Hessian matrix has both positive and negative eigenvalues. Points lying along eigenvectors associated with positive

eigenvalues have greater cost than the saddle point, while points lying along negative eigenvalues have lower value. We can think of a saddle point as being a local minimum along one cross-section of the cost function and a local maximum along another cross-section.
Critical points with high cost are far more likely to be saddle points. Critical points with extremely high cost are more likely to be local maxima.
Real neural networks also have loss functions that contain very many high-cost saddle points.
It has been shown that SGD is able to escape these areas.

8.2.4 Cliffs and Exploding Gradients:
On the face of an extremely steep cliff structure, the gradient update step can move the parameters extremely far, usually jumping off the cliff structure altogether. The cliff can be dangerous whether we approach it from above or from below, but fortunately its most serious consequences can be avoided using the gradient clipping.

8.2.5 Long-Term Dependencies: Vanishing gradients
Vanishing gradients
make it difficult to know which direction the parameters should move to improve the cost function, while exploding gradients can make learning unstable. The cliff structures described earlier that motivate gradient clipping are an example of the exploding gradient phenomenon.
Recurrent networks use the same matrix W at each time step, but feedforward networks do not, so even very deep feedforward networks can largely avoid the vanishing and exploding gradient problem.

8.2.6 Inexact Gradients:
In practice, we usually have only a noisy or even biased estimate of these quantities gradient or Hessian matrix Nearly every deep learning algorithm relies on sampling-based estimates, at least insofar as using a minibatch of training examples to compute the gradient.
In other cases, the objective function we want to minimize is actually intractable. When the objective function is intractable, typically its gradient is intractable as well. In such cases we can only approximate the gradient.
Various neural network optimization algorithms are designed to account for imperfections in the gradient estimate. One can also avoid the problem by choosing a surrogate loss function that is easier to approximate than the true loss

8.2.7 Poor Correspondence between Local and Global Structure: Poor parameter initialization

It is possible to overcome all these problems at a single point and still perform poorly if the direction that results in the most improvement locally does not point toward distant regions of much lower cost.
In this case random initialization (Gaussian or uniform) is a good strategy to break the symmetry and to allow the weights to learn independently from each other.

Explain what a reservoir computing network is. What are the main features and properties that such model owns?
Idea: fix the input to hidden and hidden to hidden connections at random values and only learn the output units' connection.
The problem of vanishing the gradient is solved because there is not backpropagation in the internal units.
Satisfy the echo state property namely, the effect of the current state h (t) and the current input x (t) on a future state h (t + tau) should vanish gradually as time passes (tau to infinity) in practice: spectral radius rue(W) < 1 namely W is contractive.
10.8 : 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 and liquid state machines. The latter is similar, except that it uses spiking neurons (with binary outputs) instead of the continuous-valued hidden units used for ESNs. Both ESNs and liquid state machines 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.
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. The training criterion may then be easily designed to be convex as a function of the output weights. For example, if the output consists of linear regression from the hidden units to the output targets, and the training criterion is

mean squared error, then it is convex and may be solved reliably with simple learning algorithms.
The important question is therefore: how do we set the input and recurrent weights so that a rich set of histories can be represented in the recurrent neural network state? The answer proposed in the reservoir computing literature is to view the recurrent net as a dynamical system, and set the input and recurrent weights such that the dynamical system is near the edge of stability.

The important question is therefore: how do we set the input and recurrent weights so that a rich set of histories can be represented in the recurrent neural network state? The answer proposed in the reservoir computing literature is to view the recurrent net as a dynamical system, and set the input and recurrent weights such that the dynamical system is near the edge of stability.
The original idea was to make the eigenvalues of the Jacobian of the state-to-state transition function be close to 1. As explained in section 8.2.5, an important characteristic of a recurrent network is the eigenvalue spectrum of the Jacobians $J(t) = \partial s(t) / \partial s(t-1)$.
Of particular importance is the spectral radius of J (t), defined to be the maximum of the absolute values of its eigenvalues. To understand the effect of the spectral radius, consider the simple case of back-propagation with a Jacobian matrix J that does not change with t. This case happens, for example, when the network is purely linear. Suppose that J has an eigenvector v with corresponding eigenvalue λ. Consider what happens as we propagate a gradient vector backward through time. If we begin with a gradient vector g, then after one step of back-propagation, we will have Jg, and after n steps we will have J^n g.
Now consider what happens if we instead back-propagate a perturbed version of g. If we begin with g + δv, then after one step, we will have J (g + δv). After n steps, we will have J^n (g + δv). From this we can see that back-propagation starting from g and back-propagation starting from g + δv diverge by δJ^n v after n steps of back-propagation. If v is chosen to be a unit eigenvector of J with eigenvalue λ, then multiplication by the Jacobian simply scales the difference at each step. The two executions of back-propagation are separated by a distance of δ|λ|^n.
When v corresponds to the largest value of |λ|, this perturbation achieves the widest possible separation of an initial perturbation of size δ. When |λ| > 1, the deviation size δ|λ|^n grows exponentially large. When |λ| < 1, the deviation size becomes exponentially small.
Of course, this example assumed that the Jacobian was the same at every time step, corresponding to a recurrent network with no nonlinearity. When a nonlinearity is present, the derivative of the nonlinearity will approach zero on many time steps and help prevent the explosion resulting from a large spectral radius. Indeed, the

most recent work on echo state networks advocates using a spectral radius much larger than unity. Everything we have said about back-propagation via repeated matrix multipli-cation applies equally to forward propagation in a network with no nonlinearity, where the state h (t+1) = h(t).T W.
When a linear map W.T always shrinks h as measured by the L^2 norm, then we say that the map is contractive.
When the spectral radius is less than one, the mapping from h (t) to h (t+1) is contractive, so a small change becomes smaller after each time step. This necessarily makes the network forget information about the past when we use a finite level of precision (such as 32-bit integers) to store the state vector. The Jacobian matrix tells us how a small change of h (t) propagates one step forward, or equivalently, how the gradient on h (t+1) propagates one step backward, during back-propagation. Note that neither W nor J need to be symmetric (although they are square and real), so they can have complex-valued eigenvalues and eigenvectors, with imaginary components corresponding to potentially oscillatory behavior (if the same Jacobian was applied iteratively). Even though h (t) or a small variation of h (t) of interest in back-propagation are real valued, they can be expressed in such a complex-valued basis. What matters is what happens to the magnitude (complex absolute value) of these possibly complex-valued basis coefficients when we multiply the matrix by the vector. An eigenvalue with magnitude greater than one corresponds to magnification (exponential growth, if applied iteratively) while a magnitude smaller than one corresponds to shrinking (exponential decay, if applied iteratively).
With a nonlinear map, the Jacobian is free to change at each step. The dynamics therefore become more complicated. It remains true, however, that a small initial variation can turn into a large variation after several steps. One difference between the purely linear case and the nonlinear case is that the use of a squashing nonlinearity such as
tanh
can cause the recurrent dynamics to become
bounded. Note that it is possible for back-propagation to retain unbounded dynamics even when forward propagation has bounded dynamics, for example, when a sequence of
tanh
units are all in the middle of their linear regime and are
connected by weight matrices with spectral radius greater than 1. Nonetheless, it is rare for all the tanh units to simultaneously lie at their linear activation point. The strategy of echo state networks is simply to fix the weights to have some spectral radius such as 3, where information is carried forward through time but does not explode because of the stabilizing effect of saturating nonlinearities like

tanh.
More recently, it has been shown that the techniques used to set the weights in ESNs could be used to initialize the weights in a fully trainable recurrent network (with the hidden-to-hidden recurrent weights trained using back-propagation through time), helping to learn long-term dependencies (Sutskever, 2012; Sutskever et al., 2013). In this setting, an initial spectral radius of 1.2 performs well, combined with the sparse initialization scheme described in section 8.4

Define a variational Autoencoder. Explain how it is used and how it can be trained.
20.10.3 Variational Autoencoders:
The variational autoencoder, or VAE, is a directed model that uses learned approximate inference and can be trained purely with gradient-based methods. To generate a sample from the model, the VAE first draws a sample z from the code distribution p model(z). The sample is then run through a differentiable generator network g(z).
Finally, x is sampled from a distribution p model (x; g(z)) =p model (x | z). During training, however, the approximate inference network (or encoder) q (z | x) is used to obtain z, and p model (x | z) is then viewed as a decoder network.

*Why is it convenient to use more than one hidden layer in neural networks? In other words, what is the advantage of multi-layer neural networks over a single hidden layer neural network?
P.195, 6.4.1:
In summary, a feedforward network with a single layer is sufficient to represent any function (universal approximation theory), but the layer may be infeasibly large and may fail to learn and generalize correctly. In many circumstances, using deeper models can reduce the number of units required to represent the desired function and can reduce the amount of generalization error.
Various families of functions can be approximated efficiently by an architecture with depth greater than some value d, but they require a much larger model if depth is restricted to be less than or equal to d. In many cases, the number of hidden units required by the shallow model is exponential in n.

Define a contractive autoencoder. Explain the differences with respect to an undercomplete autoencoder.
14.7: A contractive autoencoder is kind of overcomplete autoencoder, other kinds of overcomplete Autoencoders are Sparse and Denoising ones. The contractive autoencoder introduces an explicit regularizer on the code h = f(x), encouraging the

derivatives of, f to be as small as possible. The penalty Ω (h) is the squared Frobenius norm (sum of squared elements) of the Jacobian matrix of partial derivatives associated with the encoder function.

The summary of contractive autoencoder: Contractive autoencoder imposes a Frobenius norm on the derivatives w.r.t the inputs. This means the network privilege encodings that do not change much when the input changes, and thus are smoother. Problems of this architecture are first the heavy computation of the Jacobian for deep neural networks (so we can train one layer at a time each reconstruct the previous hidden layer autoencoder, not to overload the system). Second, the scale of the decoder should be the same as the scale of the encoder, we can do this by having the same weight matrix for both functions f and g. Just making the matrix transposed of the encoder weight matrix in the decoder weight matrix.

P 520, Second, Another practical issue is that the contraction penalty can obtain useless results if we do not impose some sort of scale on the decoder. For example, the encoder could consist of multiplying the input by a small constant epsilon, and the decoder could consist of dividing the code by epsilon. As epsilon approaches 0, the encoder drives the contractive penalty Ω (h) to approach 0 without having learned anything about the distribution. Meanwhile, the decoder maintains perfect reconstruction. This is prevented by tying the weights of, f and g. Both f and g are standard neural network layers consisting of an affine transformation followed by an element-wise nonlinearity, so it is straightforward to set the weight matrix of g to be the transpose of the weight matrix of, f.

14.1: Undercomplete Autoencoders in contrast to overcomplete Autoencoders learn an encoding of the input which is limited by architectural means (lower dimension than the input). This architecture is composed by one or more fully connected layers in decreasing size, up to the encoding layer which must be smaller than the input so the network is forced to select the most relevant features. Then, a mirror shaped dense feedforward network produces the reconstruction of the input based on the encoding in the output layer (decoder). These networks traditionally were used for dimensionality reduction and feature selection. Actually, if you have linear decoder and a MSE loss function the procedure is equal to PCA technique. Indeed, they learn to project the input data over the dimension which has the greatest variance, and therefore carries the most information of the input data.

*Define a generative adversarial network. Explain why it is useful and how it can be trained.

20.10.4: Generative adversarial networks, or GANs, are another generative modeling approach based on differentiable generator networks.

A GAN is generative model namely, it tries to estimate the true probability behind the input data, so it can sample from that probability, to extract new, synthetic samples which are as close as possible to the true samples. Unlike RBMs and VAEs, GANs use an implicit density function, they do not estimate directly the distribution under the data but sample from it indirectly.

In particular, GANs are based on a game theory approach, in that they solve a problem equivalent to a zero-sum game between two players, the generator and the discriminator. The first one tries to generate samples to fool the second one and the second has to distinguish between fake and real samples. The architecture of the GAN is composed of, initially a sample from random noise, and then a deep generator network learns to transform it in a sample which as close as possible to true samples. Then, the generated sample is fed to the discriminator and it tries to figure out if it is a true sample or a fake one. In the sample generation and evaluation part we only use the generator part of the network with a random noise input.

The equilibrium of the network is when the discriminator always outputs ½ probability. It cannot distinguish true and fake inputs. The optimization process alternatively optimizes the generator and the discriminator loss. The discriminator loss is the sum of log probability that the discriminator will spot true samples, plus the log of minus one that it will spot fake samples, this sum has to be maximized. The loss of the generator is only the second part (discriminator spots fake outputs) and it has to be minimized.

In practice, the generator optimization problem is replaced with the maximization of the log-likelihood that discriminator is wrong. This is due to performance reasons. It has been seen empirically that this loss leads to faster training and it can be explained theoretically as well. We can see in the case of generative network producing poor samples the gradient of the loss function is steeper and give us larger derivatives which lead to faster training. However, the alternation between the two optimizations makes these models hard to train. They generally produce better, less blurry samples than VAEs.

The generator aims to increase the log-probability that the discriminator makes a mistake, rather than aiming to decrease the log-probability that the discriminator makes the correct prediction. This reformulation is motivated solely by the observation that it causes the derivative of the generator's cost function with respect to the discriminator's logits to remain large even in the situation when the discriminator confidently rejects all generator samples.

***Explain in detail what the role of Monte Carlo chains in the training of stochastic neural networks is. Give an example of neural network models where Monte Carlo chains are used. ?**

To give an example of a neural network which uses MCMC (Markov chain Monte Carlo Method) we can name Restricted Boltzmann machines. RBMs are generative stochastic neural networks that can be applied to collaborative filtering techniques used by recommendation systems.

***In the context of sequential transductions, give the definition of causality and discuss how this concept is used in Recurrent Neural networks (RNN). Are all RNN architectures casual?**

?

Why do we need to use more than one layer in neural networks? In other words what is the advantage of a multi-layer neural network over a perceptron?

P 194 6.4.1:

A linear model, mapping from features to outputs via matrix multiplication, can represent only linear functions. It has the advantage of being easy to train because many loss functions result in convex optimization problems when applied to linear models. Unfortunately, we often want our systems to learn nonlinear functions. feedforward network with a linear output layer and at least one hidden layer with any "squashing" activation function (such as the logistic sigmoid activation function) can approximate any continuous function from one finite-dimensional space to another with any desired nonzero amount of error, provided that the network is given enough hidden units. The derivatives of the feedforward network can also approximate the derivatives of the function arbitrarily well.
A feedforward network with a single layer is sufficient to represent any function (universal approximation theory), but the layer may be infeasibly large and may

fail to learn and generalize correctly. In many circumstances, using deeper models can reduce the number of units required to represent the desired function and can reduce the amount of generalization error.

We need to use more than one layer in neural networks to be able to represent nonlinear functions in case of using even one hidden layer with squashing function we can represent any function (universal approximation theorem), however learning the problem is not guaranteed.

In a perceptron we can only learn linear separable problems, we need more layers and a nonlinear activation function to be able to learn nonlinear functions. In this way, it is like we are learning a different feature space, in which a linear model is able to represent the solution.

What are local minima and saddle points, and why are they a problem for optimization algorithms?

They are critical points on our optimization function.

8.2.2 Local Minima:
Local minima can be a problem if they have high cost in comparison to the global minimum. We can build small neural networks, even without hidden units, that have local minima with higher cost than the global minima.

If local minima with high cost are common, this could pose a serious problem for gradient-based optimization algorithms. ~~Whether networks of practical interest have many local minima of high cost and whether optimization algorithms encounter them remain open questions. For many years, most practitioners believed that local minima were a common problem plaguing neural network optimization. Today, that does not appear to be the case.~~
~~The problem remains an active area of research, but experts now suspect that,~~ of course, for sufficiently large neural networks, most local minima have a low cost function value, and that it is not important to find a true global minimum rather than to find a point in parameter space that has low but not minimal cost.

8.2.3 Plateaus, Saddle Points and Other Flat Regions:
another kind of point with zero gradient
Some points around a saddle point have greater cost than the saddle point, while others have a lower cost. At a saddle point, the Hessian matrix has both positive and negative eigenvalues. ~~Points lying along eigenvectors associated with positive eigenvalues have greater cost than the saddle point, while points lying along~~

~~negative eigenvalues have lower value.~~ We can think of a saddle point as being a local minimum along one cross-section of the cost function and a local maximum along another cross-section.
Critical points with high cost are far more likely to be saddle points. Critical points with extremely high cost are more likely to be local maxima.
Real neural networks also have loss functions that contain very many high-cost saddle points.
It has been shown that SGD is able to escape these areas.

What is the difference between backpropagation and (stochastic) gradient descent?
Chapter 6 and 8

When we use a feedforward neural network to accept an input x and produce an output ˆy, information flows forward through the network. The input x provides the initial information that then propagates up to the hidden units at each layer and finally produces ˆy. This is called forward propagation. During training, forward propagation can continue onward until it produces a scalar cost J (θ).
The back-propagation algorithm, often simply called backprop, allows the information from the cost to then flow backward through the network in order to compute the gradient.
The term back-propagation is often misunderstood as meaning the whole learning algorithm for multi-layer neural networks. Actually, back-propagation refers only to the method for computing the gradient, while another algorithm, such as stochastic gradient descent, is used to perform learning using this gradient.
Stochastic gradient descent (SGD) and its variants are probably the most used optimization algorithms for machine learning in general and for deep learning in particular.

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

The sparse autoencoder is an overcomplete autoencoder (the hidden code has dimension bigger than the input but with a regularization term to encourage the model to learn about the features of the input) that has a sparsity penalty of $\Omega(h)$ on the code layer h in addition to the reconstruction error, L.

The sparse autoencoder is usually trained to get some features for another task for example classification. A sparse autoencoder has to learn some statistical features of data rather than being an identity function. In this way performing the copying

of input to output will give us the result of model learning useful features of the inputs as a byproduct.

The overcomplete Autoencoders have higher dimension in h than input where in the undercomplete ones we have lower dimension in coder. The undercomplete one has limiting capacity in architecture where we limit the model by using a regularization term in overcomplete Autoencoder, so in undercomplete ones we can say that we have to discard some information.

What is a differentiable generative network? Why is it useful? Give a simple example of a Differentiable generative network. 20.10.2

Why is it useful? Many generative models are based on the idea of using differentiable generator network. The model transforms samples of latent variables z to samples x or to distributions of samples x, using a differentiable function g (z ;θ^(g)) which is typically represented by a neural network.

Examples? This class of models include variational autoencoder which use generator with an inference net and differentiable generative networks which pair this with a discriminator; and techniques to train the generator network in isolation.

What is differentiable generator networks? Generator networks are basically parameterized computational procedures to generate sample from which the architecture of the network provides the family of possible distributions to sample from and the parameters select a distribution from that family.

Why do we need the nonlinearity in neural networks? (Explain why and make an example)

To extend linear models to represent nonlinear functions we have to introduce nonlinearity in the model by transforming the input by a nonlinear transformation.

Linear models cannot understand nonlinearity, they are limited by linear functions. They also cannot interpret the interactions between any two input variables.

Example: The XOR function. When we try to learn the XOR function on a feedforward network we see that it gives out ½ everywhere in output. This happens because a linear function cannot represent the XOR function and we need some nonlinear transformation in our feature space so that a linear model could represent the solution of this problem.

Particularly, we use a feedforward network with one hidden layer and two nodes, the values computed by this hidden layer will be used as input of the next layer. The output layer is still a linear regression model but now applied to h (hidden-layer) rather than x (input). The hidden layer has a nonlinear function called an activation function.

Define Hessian and explain the ill conditioning.

Condition number is the ratio of the biggest to the smallest eigenvalue in the Hessian. It can tell us how much the second derivatives differ from one another.

Hessian matrix is the second order derivative of the function which shows us the curvature of the loss function. When we have ill conditioning of the hessian matrix, it means that the Hessian is very sensitive to changes in input data and the round off errors made during the solution procedure. It can make the SGD to be stuck meaning it would make increase the loss function a lot even when we are taking very small steps.

To find out if ill-conditioning is harmful to a neural network training task, we can monitor the squared gradient norm (g.T g) and the (g.T Hg) term. In many cases, the gradient norm does not shrink significantly throughout learning, but the g.T Hg term grows by more than 10 times w.r.t gradient norm. The result is that learning becomes very slow despite the presence of a strong gradient because the learning rate must be shrunk to compensate for even stronger curvature.

Explain why it is not a good idea to initialize all the weights of a neural network to zero. Consider both the cases when a ReLU activation function is used and the general case. Detail your answer and provide examples.

The training of NN strongly depends on initializing point, if it is bad then the learning becomes unstable. The result of the function can vary w.r.t the starting point.

The most important thing in parameter initialization is that we have to break symmetry between different units. If two hidden units with the same activation function have the same initial parameter then a deterministic learning algorithm will always update them the same way. Even if the training algorithm can use stochasticity it is better to initialize a different unit to compute a different function

from all the other units. We always like the values of the weights to be randomly initialized from a Gaussian or Uniform distribution.

In the case of the ReLU activation function we have the fact that the activation function becomes zero if we initialize the parameters to zero and we will have zero gradients, and that ReLU cannot learn from gradient based methods because of the zero gradient. Therefore it is best to set all of them to small positive values so at first all of the ReLU activations are active.

Given the definition of sequential transduction, explain the concept of memory, causality, and recursive state representation. For each different type of sequential transduction we discussed during the lectures describe a network architecture able to implement it.

The exercises were about the backpropagation on a neural networks (with ReLU), the second was about the real time recurrent learning