Deep feedforward networks, feedforward neural networks, or multilayer perceptrons
y = f*(x) maps an input x to a category y ; a feedforward network defines a mapping y = f(x ; 0) and learns the value of theta that result in the best function approximation of f*(x)

These models are feedforward since info flows through intermediate computations used to define f, and then to the output y
There are no feedback connections in which the output of the model is fed back to itself ; if feedforward NNs include feedback connections, they are recurrent neural nets

Feedforward neural nets are called networks because they are represented by the composition of many different functions , associated with a graph structure which represents a chain structure gaining more depth as the layers progress

- f^1 f^2 are the first and second layers, respectively

The final layer is the output layer
The training examples specify what the output layer must do to x - produce a value which is close to y ~ f*x
Since the training data does not show the desired output for each of the layers, they are called hidden layers

Each hidden layer in the network is vector-valued, with the dimensionality being the width of the model, and each neuron representing an increased dimension
The layer can also be thought of as consisting of many units which act in parallel as vector-to-scalar functions, since each neuron uses multiple elements from the previous layer

To extend linear models (which are normally limited) to represent nonlinear functions of x, we must transform x with phi(x) where phi is a nonlinear transformation (this can be done through the kernel trick)
Phi provides a set of features describing x or providing a new representation for x
Manually engineering phi is very complicated and requires years to learn specializations

Deep learning attempts to learn phi where we have a model
$y = f(\boldsymbol{x}; \boldsymbol{\theta}, \boldsymbol{w}) = \phi(\boldsymbol{x}; \boldsymbol{\theta})^\top \boldsymbol{w}.$ phi corresponds to the hidden layers which transform the input nonlinearly
We have parameters theta which we use to learn phi(x) from a broad class of functions
We also have parameters w which map from phi(x) to the desired output
In this approach, the representation is parametrized as phi(x ; theta) and we use the optimization algorithm to find the theta which corresponds to a good representation
This gives up the convexity of the problem, but it is generic (any # of dimensions) and it is made stronger through manually engineering in the form of choosing the right general function

Learning XOR
When only one of two values is 1, the XOR function returns 1 ; otherwise it returns 0

We want the parameter in y = f(x;theta) to be learned by our model to make f as similar to f* (the true XOR function)
We want to train our network on the points X = {[0, 0], [0, 1], [1, 0], [1, 1]}
This problem can be treated as a regression problem with an MSE loss function to simplify it (though MSE is not good in practical applications for modelling binary data)

The MSE loss function over our entire set is

$$J(\boldsymbol{\theta}) = \frac{1}{4} \sum_{\boldsymbol{x} \in \mathbb{X}} \left(f^*(\boldsymbol{x}) - f(\boldsymbol{x}; \boldsymbol{\theta})\right)^2 .$$

If we choose a linear model, with theta being broken down into w and b, our model is defined as

$$f(\boldsymbol{x}; \boldsymbol{w}, b) = \boldsymbol{x}^\top \boldsymbol{w} + b.$$

We can minimize J(theta) in closed form w/ respect to w and b using normal equations

The model learns w = 0 and b = 1/2 , and therefore outputs 0.5 everywhere
This is because our linear model is not able to represent the XOR function
We need to use a model that learns a different feature space in which a linear model works

We use a feedforward network with one hidden layer which has a vector h of hidden units computed by a function $f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c})$.
These are passed as input to the second layer, which result in the output
The output is linear, but it applies to h instead of f
The network contains two functions chained together $\boldsymbol{h} = f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c})$ and $y = f^{(2)}(\boldsymbol{h}; \boldsymbol{w}, b)$, with the final model as $f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = f^{(2)}(f^{(1)}(\boldsymbol{x}))$.

If f^1 were linear then the whole model would remain a linear function of its input
Ignoring the intercept term, we could represent our complete model function as

$$f(\boldsymbol{x}) = \boldsymbol{x}^\top \boldsymbol{w}' \text{ where } \boldsymbol{w}' = \boldsymbol{W}\boldsymbol{w}.$$

We must use a nonlinear function to describe the features, which neural networks do by using a transformation w/ learned parameters, and then applying a activation function

We do that here by defining $\boldsymbol{h} = g(\boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{c})$, where W is the weights of the linear transformation and c is the bias vector

The activation function g is chosen to be a function which is applied element wise so

$$h_i = g(\boldsymbol{x}^\top \boldsymbol{W}_{:,i} + c_i).$$

In modern neural nets, the default recommendation is the use the ReLU

$g(z) = \max\{0, z\}$

The complete network is defined as $f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = \boldsymbol{w}^\top \max\{0, \boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{c}\} + b.$

We can specify a solution to XOR now

$$\boldsymbol{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

$$\boldsymbol{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix},$$

$$\boldsymbol{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}$$

b = 0

Let x be the design matrix containing all 4 points in the binary input space

$$\boldsymbol{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}.$$

Multiply first layer weights by the input matrix to get $$\boldsymbol{XW} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix}$$

Add bias vector c to get $$\begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}.$$

In this space, all the examples lie along a line with slope of 1

As we move along this line, we need the output to start at 0, rise to 1, and drop to 0

To finish computing the value of h, apply the rectified linear transformation (ReLU) to each input

$$\begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

The points no longer lie on a simple line
Finish by multiplying by the weight vector w

$$\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Our neural network has obtained the correct answer for each example

Gradient Based Learning
Stochastic gradient descent applied to non-convex loss functions has no convergence guarantees and significantly depends on initial parameter values
Models such as linear regression and SVM can be trained with gradient descent, and this is often done with large datasets
Training a neural network is not much different from training another model in this sense
In neural nets, we must also choose a cost function and how to represent our outputs

Cost functions for neural networks tend to be the same as for other models

In mosts cases, the parametric model defines a distribution $p(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta})$ and we use the cross entropy between the model's predictions and training data as the cost function
Sometimes, instead of predicting an entire probability distribution over y, we predict some statistic of y conditioned on x (which is made easier with specialized loss functions)

Most neural networks are trained using max likelihood, meaning that the cost function is the negative log likelihood, or cross entropy defined by

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{\text{data}}} \log p_{\text{model}}(\boldsymbol{y} \mid \boldsymbol{x}).$$

The expected value of the log of the model's probability given x and y from the true data distribution
If $p_{\text{model}}(\boldsymbol{y} \mid \boldsymbol{x}) = \mathcal{N}(\boldsymbol{y}; f(\boldsymbol{x}; \boldsymbol{\theta}), \boldsymbol{I})$ then we expand the cost function and disregard terms which aren't dependent on the model parameters to recover the MSE cost

$$J(\theta) = \frac{1}{2}\mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{\text{data}}} ||\boldsymbol{y} - f(\boldsymbol{x}; \boldsymbol{\theta})||^2 + \text{const.}$$

The disregarded constant is based on the variance of the distribution which we chose not to parameterize with theta

Regardless of the f(x;theta) - whether or not it is a linear model - used to predict the mean of the Gaussian, the equivalence between max likelihood estimation and MSE to find the best model holds
Specifying a model p(y | x) automatically determiens a cost function log p(y | x)

The gradient of the cost function must be large enough to serve as a good guide for the learning algorithm ; often, activation functions will saturate and cause the cost function to become flat
Negative log likelihood avoids this problem since the log function undoes the exp which would otherwise cause very negative values to saturate

Cross entropy does not have a minimum when applied to models in practice ; for discrete outputs / classification, the models cannot actually represent a probability of 0 or 1 but they can come arbitrarily close to doing so (e.g logistic regression)
If the model can control the density of the output distribution, then it is possible to assign very high density to the correct training set outputs, resulting in a cross entropy approaching negative infinity
Regularization techniques can prevent a model from abusing this

Instead of learning a full probability distribution p(y | x;theta) we want to sometimes learn one conditional statistic of y given x
For example, we may just want to predict the mean of y (rather than each possible state)

If we have a powerful neural net, we can think of it as being able to represent any class f from a wide class of functions, only limited by continuity and boundedness (not by parameterization)
The cost function is functional - a mapping from functions to real numbers
We can design the cost function to have its minimum occur at some specific function we choose
We can design the cost functional to have its minimum lie on the function that maps x to the expected value of y given x and then optimize to find the function which satisfies this
Solving an optimization problem with respect to a function can be done through calculus of variations, which helps us derive two key results

Solving the optimization problem $$f^* = \arg\min_f \mathbb{E}_{\mathbf{x},\mathbf{y}\sim p_{\text{data}}} ||\boldsymbol{y} - f(\boldsymbol{x})||^2$$

yields $f^*(\boldsymbol{x}) = \mathbb{E}_{\mathbf{y}\sim p_{\text{data}}(\boldsymbol{y}|\boldsymbol{x})}[\boldsymbol{y}]$, which predicts the mean of y for each value of x

A second result derived using calculus of variations is that

$$f^* = \arg\min_f \mathbb{E}_{\mathbf{x},\mathbf{y}\sim p_{\text{data}}} ||\boldsymbol{y} - f(\boldsymbol{x})||_1$$

yields a function that predicts the median value of y for each value of x, as long as a function is described in the family of functions optimize over
This is known as mean absolute error

Cross entropy is typically more popular than MSE or MAE, even when you don't need to estimate an entire distribution, due to saturating outputs

Most of the time, we use cross entropy between data and model distribution as the cost function
Deciding how to represent the output determines the form of the cross entropy function

One simple output is an output unit based on a transformation with no nonlinearity
There are called linear units

Given features h, a layer of linear outputs produces an output a vector $\hat{\boldsymbol{y}} = \boldsymbol{W}^\top \boldsymbol{h} + \boldsymbol{b}$
Linear output layers are used to produce the mean of a conditional Gaussian distribution
$p(\boldsymbol{y} \mid \boldsymbol{x}) = \mathcal{N}(\boldsymbol{y}; \hat{\boldsymbol{y}}, \boldsymbol{I})$.
Maximize the log likelihood is therefore equivalent to minimizing MSE
Max log likelihood makes it easy to learn the covariance of the Gaussian or make the covariance a function of the input, if the covariance is constrained to be positive definite matrix for all inputs
It is difficult to satisfy this constraint with a linear output layer, so other output units are used to parametrize the covariance
Linear units don't saturate, so they are good for gradient based learning

The max likelihood approach of predicting a binary variable y is to define a Bernoulli distribution over y conditioned on x
A Bernoulli distribution is defined by just a single number, and the only thing that needs to be predicted is $P(y = 1 \mid \boldsymbol{x})$. which is in range [0, 1]

This constraint requires careful design effort
Suppose we were to use a linear unit and threshold its value to obtain a probability

$$P(y = 1 \mid \boldsymbol{x}) = \max\left\{0, \min\left\{1, \boldsymbol{w}^\top \boldsymbol{h} + b\right\}\right\}$$

This would be problematic because if $w^T h + b$ went outside the unit interval, the gradient of the output of the model would be 0, which means the learning algorithm has no guide for improving the parameters
It is instead better to use a different approach which ensures there is a stronger gradient when the model has the wrong answer, which can be done by combining max likelihood w/ sigmoid
The sigmoid is defined as: $\hat{y} = \sigma\left(\boldsymbol{w}^\top \boldsymbol{h} + b\right)$ where sigma is the logistic sigmoid function

Sigmoid output unit has 2 components: a linear layer which computes $w^T h + b$, and a sigmoid activation to convert z into a probability

Sigmoid can be motivated by constructing an unnormalized probabiltiy distirbution P(y) which doesn't sum to 1
We divide by an appropriate constant to obtain a valid probability distribution
If the unnormalized log probabilities are linear in y and z, we exponentiate to obtain the unnormalized probabilities
We normalize to see this yields a Bernoulli distribution controlled by a sigmoidal transformation of z

$$\begin{aligned}\log \tilde{P}(y) &= yz \\ \tilde{P}(y) &= \exp(yz) \\ P(y) &= \frac{\exp(yz)}{\sum_{y'=0}^{1} \exp(y'z)} \\ P(y) &= \sigma\left((2y-1)z\right).\end{aligned}$$

The z variable defining a distribution based on exponential and normalization over binary variables is called a logit

The approach to predicting probabilities in log space is natural to use with max likelihood
Since the cost function used w/ max likelihood is -log P(y | x), the log in the cost function undoes the exp of the sigmoid
Without this, the saturation of the sigmoid could prevent gradient based learning from making good progress

The loss function for max likelihood learning of a Bernoulli distribution is

$$\begin{aligned}J(\boldsymbol{\theta}) &= -\log P(y \mid \boldsymbol{x}) \\ &= -\log \sigma\left((2y-1)z\right) \\ &= \zeta\left((1-2y)z\right).\end{aligned}$$

By re-writing in terms of softplus, can see that it saturates only when (1-2y)(z) is very negative
Saturation only occurs when the model has the right answer - when y = 1 and z is very positive (as it should be), or y = 0 and z is very negative

When z has the wrong sign, the argument to the softplus function may be simplified to $|z|$.
As $|z|$. becomes large while z has the wrong sign, the softplus asymptotes towards returning its argument $|z|$.
The derivative w/ respect to z asymptotes to sign(z) so in the presence of an incorrect z, the softplus function does not shrink the gradient, and gradient-based learning can act quickly to correct a mistaken z

When we use other loss functions such as MSE, the loss can saturate whenever $\sigma(z)$ saturates
The sigmoid activation saturates to 0 when z is very negative and saturates to 1 when z is very positive, which causes the gradient to be too small for learning
This is why max likelihood is the preferred approach

In software implementation, to avoid numerical problems, negative log likelihood should be written as a function of z rather than as a function of $\hat{y} = \sigma(z)$. since if the sigmoid function underflows to zero, then taking the logarithm of y yields negative infinity

Sigmoid assigns probability of x = 1 for each value of x
If there are more than 2 discrete states, we use softmax

- Can be used to choose between multiple states for a model's internal variable

For binary variables, we chose to predict $z = \log \tilde{P}(y = 1 \mid \boldsymbol{x})$. since we want the log to be well-behaved for log likelihood optimization, and because it needs to lie between 0 and 1
Exponentiation and normalization gave a Bernoulli distribution controlled by sigmoid

We want to produce a vector y | $\hat{y}_i = P(y = i \mid \boldsymbol{x})$. with the vector summing to 1
Similar to sigmoid, a linear layer predicts unnormalized log probabilities

$$\boldsymbol{z} = \boldsymbol{W}^\top \boldsymbol{h} + \boldsymbol{b},$$

$$z_i = \log \tilde{P}(y = i \mid \boldsymbol{x})$$

Softmax can exponentiate and normalize z to obtain the predicted y

$$\text{softmax}(\boldsymbol{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}.$$

Maximizing log P(y = *i* ; z) => maximizes log softmax(z)$_i$
Defining softmax in terms of exp is natural b/c log in log-likelihood undoes exp of softmax

$$\log \text{softmax}(\boldsymbol{z})_i = z_i - \log \sum_j \exp(z_j).$$

Since z_i contributes to the loss, the term cannot saturate
The first term encourages z_i to be pushed up while the second term encourages all of z to be pushed down
The second term can roughly be approx. by $\max_j z_j$ since $\exp(z_k)$ is insignificant if z_k is noticeably less than z_j

Negative log likelihood therefore always penalizes the most active incorrect prediction ; if the prediction is correct, the first and second term cancels out and the example barely contributes

Unregularized max likelihood drives the model to learn parameters that drive softmax to predict fractions of counts of each outcomes observed in the training set

$$\text{softmax}(\boldsymbol{z}(\boldsymbol{x};\boldsymbol{\theta}))_i \approx \frac{\sum_{j=1}^{m} \mathbf{1}_{y^{(j)}=i,\boldsymbol{x}^{(j)}=\boldsymbol{x}}}{\sum_{j=1}^{m} \mathbf{1}_{\boldsymbol{x}^{(j)}=\boldsymbol{x}}}.$$

Since max likelihood is a consistent estimator, this is guaranteed to happen if the model family can represent the training distribution
Many objective functions don't work well with softmax, since they do not undo the exp causing the softmax's gradient to vanish when its argument is negative

When the values between the inputs are significantly different for softmax, the output values can saturate, causing its cost function to saturate unless they can invert it
Softmax is invariant to adding the same scalar to its input as it only considers their difference:

$$\text{softmax}(\boldsymbol{z}) = \text{softmax}(\boldsymbol{z} + c).$$

We can derive a numerically stable variant of softmax

$$\text{softmax}(\boldsymbol{z}) = \text{softmax}(\boldsymbol{z} - \max_i z_i).$$

This reformulation has very small numerical errors
The softmax function is driven by the amount that its arguments deviate from $\max_i z_i$

An output $\text{softmax}(z)_i$ saturates to 1 when its corresponding input is max $(z_i = \max_i z_i)$ and z_i is much greater than all other inputs ; the other inputs saturate to 0
The loss function must compensate for this

The most common way for z to be produced is to have a linear layer, but this overparametrizes the distribution
The constraint n outputs must sum to 1 means that only n - 1 outputs are necessary
We can impose a requirement that one element of z is fixed: $z_n = 0$, which sigmoid does

Defining $P(y = 1 \mid \boldsymbol{x}) = \sigma(z)$ is equivalent to defining $P(y = 1 \mid \boldsymbol{x}) = \text{softmax}(\boldsymbol{z})_1$ with a 2 dim z and $z_1 = 0$
The n - 1 argument and n argument have different learning dynamics, but there is rarely a difference in practice
Softmax can be thought of as winner take all, since the activation of one neuron causes deactivation of others
Soft represents differentiability, and argmax has its results OHE so its not differentiable

- Softmax is therefore a "soft" version of argmax

Max likelihood principle gives guidelines for defining a good cost function for any output layer

If we define a conditional distribution p(y | x ; theta), we use $-\log p(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta})$ as our cost
The neural network function f is not a direct prediction of y ; rather, it provides the parameters for a distribution over y $f(\boldsymbol{x};\boldsymbol{\theta}) = \boldsymbol{\omega}$
The loss is $-\log p(\mathbf{y}; \boldsymbol{\omega}(\boldsymbol{x}))$.

Suppose we want to learn the variance of a conditional Gaussian for y, given x

- It is simply the empirical mean of the squared difference between observations and their expected value => there is a closed form solution
- Include the variance as one of the properties of $p(\mathbf{y} \mid \boldsymbol{x})$ which is controlled by $\boldsymbol{\omega} = f(\boldsymbol{x}; \boldsymbol{\theta})$
- Then the negative log likelihood $-\log p(\boldsymbol{y}; \boldsymbol{\omega}(\boldsymbol{x}))$ provides a cost function which makes our optimization procedure learn the variance
- When the standard deviation does not depend on the input, we can introduce a new parameter in the network, copied directly into w, such as Beta = 1/(sigma^2)

A heteroscedastic model predicts different amount of variance in y for different values of x
If heteroscedastic, we make the variance be one of the outputs of f(x;theta) which can be done by formulating the distribution using precision
We use a diagonal precision matrix $\mathrm{diag}(\boldsymbol{\beta})$. in the multivariate case
This only involves multiplication by B$_i$ and addition of log B$_i$ which works well w/ grad. descent
If we parametrized in terms of variance or standard deviation, this can result in very large gradients or vanishing gradients
Learn more about precision

We must ensure the covariance matrix of the Gaussian is positive definite, and since the eigenvalues of the precision matrix are reciprocals of covariance, this ensures the precision matrix is positive definite
If we use a diagonal matrix, the output must be positive
If a is the raw activation used to determine diagonal precision, we can use softplus to obtain a positive precision vector $\boldsymbol{\beta} = \zeta(\boldsymbol{a})$.

If the covariance **Σ** is full and conditional, parametrization must be chosen which guarantees positive-definiteness of predicted covariance matrix
This can be done with $\boldsymbol{\Sigma}(\boldsymbol{x}) = \boldsymbol{B}(\boldsymbol{x})\boldsymbol{B}^{\top}(\boldsymbol{x})$, where B is an unconstrained square matrix; if the matrix is full rank, this will require O(d^3) for the eigendecomposition

Multimodal regression - predict real values that come from a conditional distribution p(y | x) that can have several peaks in y space for the same x

Gaussian mixture is a natural representation of this; mixture density networks output mixtures

Gaussian mixture w/ n components: $p(\boldsymbol{y} \mid \boldsymbol{x}) = \sum_{i=1}^{n} p(\mathrm{c} = i \mid \boldsymbol{x}) \mathcal{N}(\boldsymbol{y}; \boldsymbol{\mu}^{(i)}(\boldsymbol{x}), \boldsymbol{\Sigma}^{(i)}(\boldsymbol{x})).$ where c is a latent variable

The neural network must output a vector defining p(c = *i* | x), a matrix providing mu^i(x) for each i, and a tensor providing $\boldsymbol{\Sigma}^{(i)}(\boldsymbol{x})$ for each i (the network is outputting a model here)

Mixture components p(c = i | x) form a multinoulli distribution over the n different components associated with c (the random component)

- These are obtained by a softmax over an n-dimensional vector to ensure the probabilities are positive and sum to 1
- Means $\boldsymbol{\mu}^i$(x) indicate the mean x associated with the i-th Gaussian component, unconstrained typically with no nonlinearity
- If y is a d-vector, the network outputs an n by d matrix containing all n of the d-dimensional vectors
- In practice, we do not know which component produced each observation; negative log likelihood weighs each example's contribution to the loss by the probability that the component produced the example
- Covariance $\boldsymbol{\Sigma}^i$(x) specify covariance matrix for each component i
- Similar to learning a single Gaussian component, we use a diagonal matrix to avoid determinant computations
- Max likelihood is complicated in the same way for covariance as the mean; if correct specification of negative log likelihood is given, grad. descent follows this process

Gradient based optimization of conditional Gaussian mixtures can be unreliable since divisions by the variance can be numerically unstable (variance becomes very small => large gradients)
One solution is to clip gradients
Mixture density strategy lets the network represent multiple output modes and control output variance, which is crucial for practical applications

ReLUs are the default choice of hidden units
Not differentiable at many inputs, however (such as 0 or below for a ReLU)
Since we do not actually expect the model tor each a gradient of 0, software implementations return one-sided derivatives to get past non-differentiability of certain points of hidden unit activation functions
Hidden units are distinguished by their choice of activation g(z)

Gradients of a RELU are large and consistent for half its domain
Since the second derivative is 0, gradients of a ReLU are far more useful for learning
ReLUs are put on top of an affine transformation $\boldsymbol{h} = g(\boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{b}).$; it is good practice to initialize b to a small value so that ReLUs will be initially active

Some varieties of ReLUs ensure they have gradients everywhere, so they can always learn

$z_i < 0$: $h_i = g(\boldsymbol{z}, \boldsymbol{\alpha})_i = \max(0, z_i) + \alpha_i \min(0, z_i)$

Absolute value rectification: alpha = -1, so g(z) = |z|

- Used in image recognition, to seek features which are invariant to a polarity reversal of input illumination (neurons activating)

Leaky ReLU fixed alpha to a small value, while a parameter ReLU treats alpha asl earnable

Maxout units, rather than applying an element-wise function g(z), divide z into groups of k values

Each maxout unit then outputs the maximum element of one of these groups

$g(\boldsymbol{z})_i = \max_{j \in \mathbb{G}^{(i)}} z_j$ where G^i is the set of indices corresponding to group i, {(i-1)k + 1… ik}

This provides a way of learning a piecewise linear function responding to multiple directions in the input x space

Maxout unit can learn a piecewise linear, convex function with up to k pieces; it can be seen as learning the activation function itself, rather than relationshihps between units

With a large enough k, a maxout unit can approximate any convex function; a maxout layer w/ 2 pieces can implement the same function of the input x as a traditional layer using ReLU

Each maxout unit is parametrized by k weight vectors, instead of 1, so regularization is needed

If the features captured by n different linear filters can be summarized w/o losing info by taking the max over each group of k features, the next layer can get by w/ k times fewer weights

- This is the idea behind convolutional neural networks

Since each unit is driven by multiple filters, maxout units have redundancy which help them avoid catastrophic forgetting

ReLUs and generalizations are based on the principle that models are easier to optimize if their behavior is linear; this applies when dealing with RNNs as well

Prior to ReLUs, the logistic sigmoid $g(z) = \sigma(z)$ or the hyperbolic tangent was used $g(z) = \tanh(z)$.

Recall sigmoid units saturate which makes gradient based learning hard

They are useful as output units if an appropriate cost function can undo their saturation

tanh and logistic sigmoid are similar, but tanh is usually better since it resembles the identity function more closely: tanh(0) = 0, while sig(0) = ½

Since tanh is similar to the identity function near 0, training a network $\hat{y} = \boldsymbol{w}^\top \tanh(\boldsymbol{U}^\top \tanh(\boldsymbol{V}^\top \boldsymbol{x}))$ resembles training a linear model y = $\boldsymbol{w}^\top \boldsymbol{U}^\top \boldsymbol{V}^\top \boldsymbol{x}$ if the network activations are kept small, which makes training it easier

Sigmoidal activations are good for probabilistic models and RNNs

New hidden unit types which perform just as well as ReLUs are common
Not having an activation can be considered as using the identity function as the activation
It is acceptable for some layers to be purely linear
Consider a network layer with n inputs and p outputs $\boldsymbol{h} = g(\boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{b})$.
We replace this with two layers, with one layer using weight matrix U and the other using weight matrix V; if the first layer has no activation, then we factored the weight matrix of the original layer based on W
The factored approach is to compute $\boldsymbol{h} = g(\boldsymbol{V}^\top \boldsymbol{U}^\top \boldsymbol{x} + \boldsymbol{b})$
If U produces q outputs, U and V together contain (n + p)q parameters, while W contains np parameters
For a small q, this can be computationally efficient; this comes at the cost of constraining the linear transformation to be low rank (less representation), but these are often sufficient
Softmax units are sometimes used in hidden layers, when trying to manipulate memory

RBF unit: $h_i = \exp\left(-\frac{1}{\sigma_i^2}||\boldsymbol{W}_{:,i} - \boldsymbol{x}||^2\right)$. becomes more active as x approaches a template W; it can be difficult to optimize since it saturates to 0 for most x
Softplus: $g(a) = \zeta(a) = \log(1+e^a)$. ; smooth version of rectifier, differentiable everywhere, and saturates less, but empirically not that good
Hard tanh: shaped similarly to tanh and the rectifier, but it is bounded, g(a) = max(-1, min(1,a))

Architecture Unit
Architecture refers to overall structure of the network - # of units and how they're connected
Layers are arranged in a chain structure

$$\boldsymbol{h}^{(1)} = g^{(1)}\left(\boldsymbol{W}^{(1)\top}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right)$$

$$\boldsymbol{h}^{(2)} = g^{(2)}\left(\boldsymbol{W}^{(2)\top}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)}\right)$$

Deeper networks use far fewer units per layer and fewer parameters, but are hard to optimize

A linear model mapping from features to outputs using matrices can only represent linear functions; however, it is easy to train since loss functions result in convex optimization problems when applied to linear models
Universal approximation theorem - feedforward network w/ linear output layer and at least one hidden layer w/ a squashing activation function (like sigmoid) can approximate a Borel measurable function w/ any nonzero error, provided it has enough units
The derivatives of the feedforward network can approx. the derivatives of the Borel function
A borel function is defined as a continuous function on a closed, bounded subset of R^N
A neural network can approximate a mapping between finite dimensional discrete spaces

A large MLP can represent this function, but the training algorithm may not be able to learn it

The optimization algorithm may not be able to find the correct parameters, or the training algorithm might overfit
In the worst case, an exponential number of hidden units (corresponding to each different input configuration) may be required
$v \in \{0, 1\}^n$ The # of possible binary functions is $2^{2^{\wedge}n}$ and selecting a function requires 2^n which requires $O(2^n)$ degrees of freedom

Some models can be approximated by an architecture w/ depth greater than d, but require a larger model if depth is restricted to be less than d
Number of hidden units required by the shallow model is exponential in n
A sufficiently wide rectifier network can represent any function, but there's no guide on # of units

Piecewise linear networks (which can be obtained from recitfier nonlinearities / maxout units) can represent functions with a number of regions exponential in the depth of the network
Adding extra layers composes operations such that we obtain an exponentially large # of piecewise linear regions which can capture all kinds of patterns

The number of linear regions carved out by a deep rectifier network w/ d inputs, depth l, and n units per hidden layer is $O\left(\binom{n}{d}^{d(l-1)} n^d\right),$, which is exponential in the depth l

If using maxout networks w/ k filters per unit, number of linear regions is $O\left(k^{(l-1)+d}\right).$

Using a deep network is essentially having a prior which says we want to learn a function composed of several simpler functions; the learning problem discovers a set of underlying factors of variation, which can be described in terms of other, simpler factors of variations (the inputs)

In the default neural net, every input unit is connected to every output unit
Specialized networks have fewer connections

Backpropagation and Other Differentiation Algorithms
Forward prop continues until it produces a scalar cost J(theta)
Backprop lets this info flow backwards through the network to compute the gradient
Backprop efficiently calculates an expression for the gradient
Backprop is just the means for computing the gradient, not the learning algorithm itself
Backprop is not just for multi-layer networks

$\nabla_{x} f(x, y)$ x is a set of variables whose derivatives are desired, y has inputs to the function, but its derivatives are not required

We want the gradient of the cost function with respect to the parameters: $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$.
Backprop can compute values such as the Jacobian of a function w/ multiple outputs
For our purposes, we consider f to have a single output

Each node in a graph indicates a scalar, tensor, matrix, or vector
We have operations, a function of one or more variable; multiple simple operations form a more complex one
An operation returns only a single output, which may be a vector with multiple entries
An edge from x to y indicates an operation was performed going between the two; an operation may be written near the output node y

Chain rule of calculus computes derivatives of functions formed by composing other functions whose derivatives are known
y = g(x), z = f(g(x)) = f(y) => $\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$.
We can generalize beyond scalars: x is R^m and y R^n, g maps from R^m to R^n, and f maps from R^n to R
If y = g(x) and z = f(y)

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

This is expressed in vector notation as $\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^\top \nabla_{\boldsymbol{y}} z,$ where dy/dx is the n x m Jacobian matrix of z
Backprop performs a Jacobian-gradient product (as shown above) for each op in the graph

To denote the gradient of a value z w/ respect to a tensor X, we write $\nabla_{\mathbf{X}} z$ as if X were a vector
Indices into X have multiple coordinates: a 3-D tensor is indexed by 3 coorsd
Use a variable i to represent the complete tuple of indices, abstracting these coords away
$(\nabla_{\mathbf{X}} z)_i$ gives $\frac{\partial z}{\partial \mathsf{X}_i}$ for all possible integer indices i, just as how all possible indices into a vector x $(\nabla_{\boldsymbol{x}} z)_i$ gives $\frac{\partial z}{\partial \mathsf{X}_i}$.

If Y = g(X) and z = f(Y) where X and Y are tensors,

$$\nabla_{\mathbf{X}} z = \sum_j (\nabla_{\mathbf{X}} \mathsf{Y}_j) \frac{\partial z}{\partial \mathsf{Y}_j}.$$

Many subexpressions may be repeated several times within the overall expression for the gradient

When computing the gradient, the procedure needs to be efficient, without wasting resources re-computing the same gradient with a naive chain rule implementation

- Computing the same gradient twice may be more efficient in terms of memory

Begin by a version of the backprop algorithm which specifies the gradient computation directly in the order it will be done and according to the recursive app of the chain rule
Algorithm can be considered as a symbolic specification of the computational graph for computing backprop; however, this does not make explicit the construction of a symbolic graph

A computation graph which calculates $u^{(n)}$ the scalar loss

We want to obtain the gradient w/ respect to the n_i input nodes $u^{(1)}$ to $u^{(n_i)}$; we want to compute $\frac{\partial u^{(n)}}{\partial u^{(i)}}$ for all $i \in \{1, 2, \ldots, n_i\}$
$u^1 \ldots u^{ni}$ correspond to parameters of the model, while u^n corresponds to the loss
Assume the nodes have been ordered in such a way we can compute their output, starting at u^{ni+1} and going up to u^n

Each node u^i is associated with f^i and is computed by evaluating $u^{(i)} = f(\mathbb{A}^{(i)})$ where A^i is the set of all parent nodes of u^i
This specifies forward prop

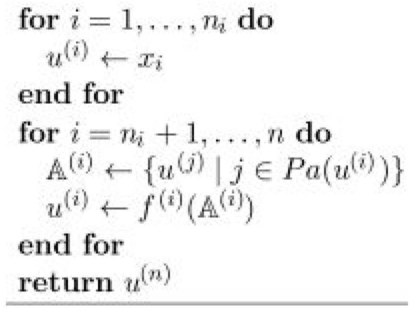
To perform backprop, we construct a computational graph which depends on $\mathcal{G}$ and adds it to an extra set of nodes, which form a graph subgraph B with one node per node of G
Computation in B proceeds in the reverse order of computation in G, and each node computes the derivative $\frac{\partial u^{(n)}}{\partial u^{(i)}}$ associated with the forward graph node u^i
This is done using the chain rule w/ using the scalar output u^n

$$\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:j \in Pa(u^{(i)})} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}}$$

B contains one edge for each edge from node u^j to u^i of G which is associated with the computation of $\frac{\partial u^{(i)}}{\partial u^{(j)}}$.
A dot product is performed, between the gradient computed w/ respect to u^i which are children of u^j and the vectors containing the partial derivatives $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ for the same children nodes
The amount of computation required for performing backprop scales linearly w/ the number of edges in G, where the computation for each edge consists of computing a partial derivative of a node w/ respect to its parents, and a multadd
Tensor-valued nodes groups multiple scalar values in the same node and enables more efficient implementations

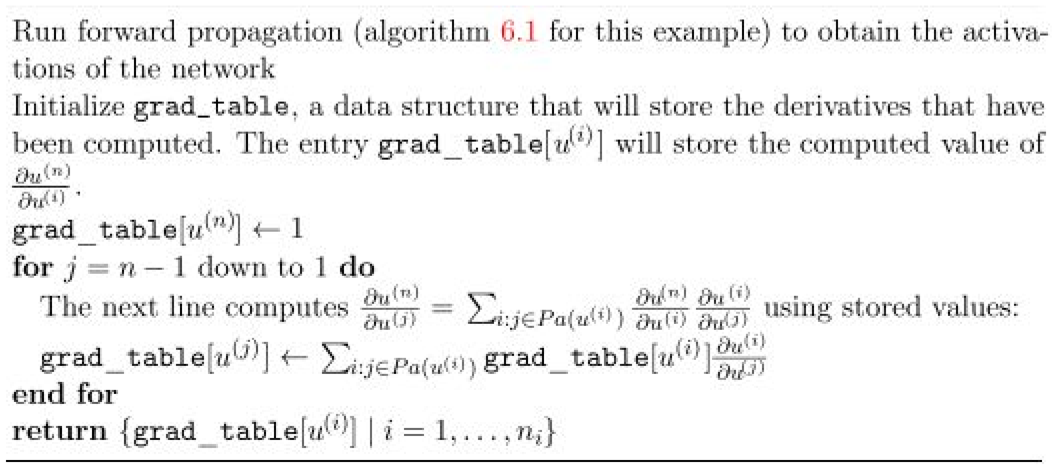
Backprop visits each edge from u^j to u^i of a graph exactly once to obtain the associated partial derivative du^i / du^j; therefore it performs on the order of one Jacobian product per graph node
Backprop avoids exponential explosion in repeated subexpressions; other algorithms can be more efficient by recomputing which allows memory conservation, and by simplifying the computational graph

6.1



```
for $i = 1, \ldots, n_i$ do
    $u^{(i)} \leftarrow x_i$
end for
for $i = n_i + 1, \ldots, n$ do
    $\mathbb{A}^{(i)} \leftarrow \{u^{(j)} \mid j \in Pa(u^{(i)})\}$
    $u^{(i)} \leftarrow f^{(i)}(\mathbb{A}^{(i)})$
end for
return $u^{(n)}$
```

u^i to u^{ni} are the inputs; j is a parent node of u^i



```
Run forward propagation (algorithm 6.1 for this example) to obtain the activations of the network
Initialize grad_table, a data structure that will store the derivatives that have been computed. The entry grad_table[$u^{(i)}$] will store the computed value of $\frac{\partial u^{(n)}}{\partial u^{(i)}}$.
grad_table[$u^{(n)}$] $\leftarrow 1$
for $j = n - 1$ down to 1 do
    The next line computes $\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:j \in Pa(u^{(i)})} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}}$ using stored values:
    grad_table[$u^{(j)}$] $\leftarrow \sum_{i:j \in Pa(u^{(i)})}$ grad_table[$u^{(i)}$]$\frac{\partial u^{(i)}}{\partial u^{(j)}}$
end for
return {grad_table[$u^{(i)}$] $\mid i = 1, \ldots, n_i$}
```

Pa means parent, and grad_table is used to recursively compute partial derivatives

Algorithm 6.3 is for forward prop in a specific graph associated w/ a MLP

Algorithm 6.3 Forward propagation through a typical deep neural network and the computation of the cost function. The loss $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$ depends on the output $\hat{\boldsymbol{y}}$ and on the target $\boldsymbol{y}$ (see section 6.2.1.1 for examples of loss functions). To obtain the total cost J, the loss may be added to a regularizer $\Omega(\theta)$, where θ contains all the parameters (weights and biases). Algorithm 6.4 shows how to compute gradients of J with respect to parameters $\boldsymbol{W}$ and $\boldsymbol{b}$. For simplicity, this demonstration uses only a single input example $\boldsymbol{x}$. Practical applications should use a minibatch. See section 6.5.7 for a more realistic demonstration.

```
**Require:** Network depth, $l$
**Require:** $\boldsymbol{W}^{(i)}, i \in \{1, \ldots, l\}$, the weight matrices of the model
**Require:** $\boldsymbol{b}^{(i)}, i \in \{1, \ldots, l\}$, the bias parameters of the model
**Require:** $\boldsymbol{x}$, the input to process
**Require:** $\boldsymbol{y}$, the target output
  $\boldsymbol{h}^{(0)} = \boldsymbol{x}$
  for $k = 1, \ldots, l$ do
    $\boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)}\boldsymbol{h}^{(k-1)}$
    $\boldsymbol{h}^{(k)} = f(\boldsymbol{a}^{(k)})$
  end for
  $\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)}$
  $J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda\Omega(\theta)$
```

Algorithm 6.4 Backward computation for the deep neural network of algorithm 6.3, which uses in addition to the input $\boldsymbol{x}$ a target $\boldsymbol{y}$. This computation yields the gradients on the activations $\boldsymbol{a}^{(k)}$ for each layer k, starting from the output layer and going backwards to the first hidden layer. From these gradients, which can be interpreted as an indication of how each layer's output should change to reduce error, one can obtain the gradient on the parameters of each layer. The gradients on weights and biases can be immediately used as part of a stochastic gradient update (performing the update right after the gradients have been computed) or used with other gradient-based optimization methods.

```
After the forward computation, compute the gradient on the output layer:
$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{y})$
for $k = l, l-1, \ldots, 1$ do
  Convert the gradient on the layer's output into a gradient into the pre-
  nonlinearity activation (element-wise multiplication if $f$ is element-wise):
  $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$
  Compute gradients on weights and biases (including the regularization term,
  where needed):
  $\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega(\theta)$
  $\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g}\, \boldsymbol{h}^{(k-1)\top} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Omega(\theta)$
  Propagate the gradients w.r.t. the next lower-level hidden layer's activations:
  $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)\top}\, \boldsymbol{g}$
end for
```

We decompose the gradient from the layer's output into the activation and then to the biases and weights

Computational graphs evaluate on symbols - variables which do not have specific values
These algebraic / graph based representations are called symbolic representations
We replace a symbolic input to the network x with a specific numeric value (vector)

Symbol-to-number differntition: take a computational graph and set of numerical values for the inputs to the graph, and return a set of values describing the gradients at those inputs
Alternatively, take a computational graph and add additional nodes which provide a symbolic description of the desired derivatives; the derivatives use the same terms as the original neural net function f
Derivatives are part of the graph => they can be further differntiated to obtain high order ones

The latter approach is used; subsets of the graphs can be computed using specific values
A generic graph evaluation engine can evaluate every node as soon as its parents' values are available

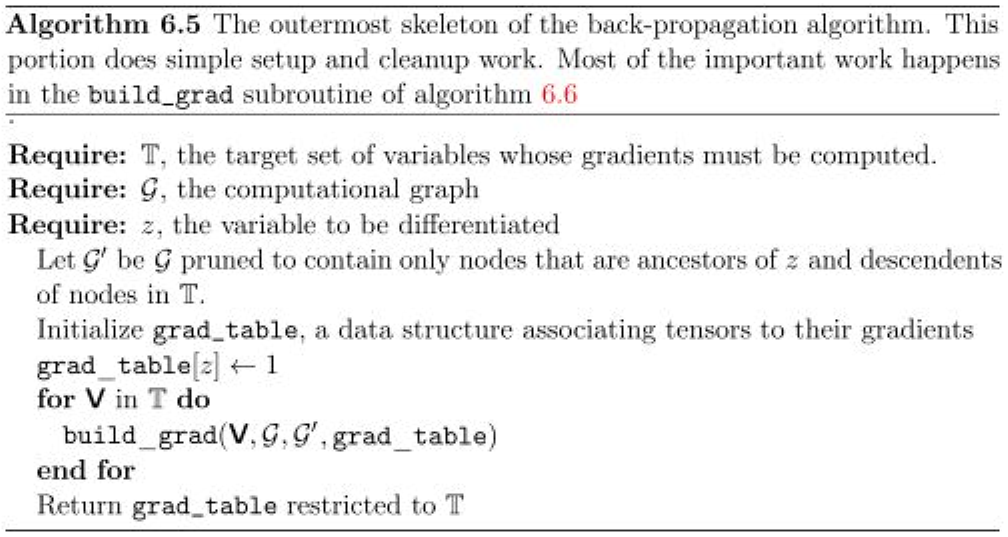
dz/dz = 1
Compute the gradient w/ respect to each parent of z by multiplying the current gradient by the Jacobian of the operation that produced z
Keep on multiplying by Jacobians travelling backwards until we reach x
If a node is reached by two or more paths, sum the gradients arriving from different paths

Each node in the graph represents a variable, which we describe as a tensor **V** (tensors are generalized to have any dimensions such as that of a matrix, vector, scalar, etc)
Assume each variable V is associated w/ the following:

- get_operation(V) returns the operation which computes V (e.g matrix multiplication)
- get_consumers(V, G) returns a list of children of V in the graph G
- get_inputs(V, G) returns a list of parents of V in the graph G

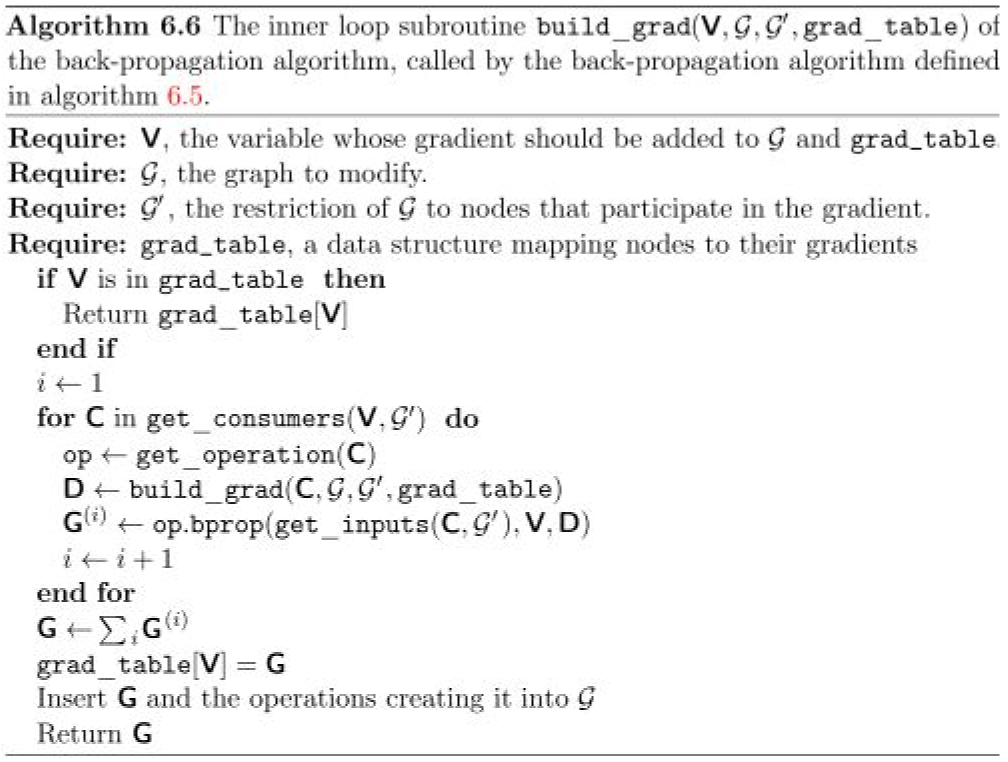
Each operation op is associated with a bprop operation, which can compute a Jacobian vector product; each operation must know how to backpropagate through its edges in the graph
For example, if C = AB, and we know the gradient of the output is G, the gradient w/ respect to A is give by GB^T
Backprop must call each operation's bprop rules with the right arguments

op.bprop(inputs, X, G) must return $\sum_i (\nabla_{\mathbf{X}} \texttt{op.f(inputs)}_i)\, G_i,$
Inputs is a list of inputs supplied to the operation, op.f is the function the op implements, X is the input whose gradient we wish to compute, and G is the gradient of the entire op's outptu
op.bprop should always pretend all of its inputs are distinct; derivative of x * x (x^2) should return x w/ respect to each input, which will later be summed up to 2x



Algorithm 6.5 The outermost skeleton of the back-propagation algorithm. This portion does simple setup and cleanup work. Most of the important work happens in the `build_grad` subroutine of algorithm 6.6

```
Require: $\mathbb{T}$, the target set of variables whose gradients must be computed.
Require: $\mathcal{G}$, the computational graph
Require: $z$, the variable to be differentiated
  Let $\mathcal{G}'$ be $\mathcal{G}$ pruned to contain only nodes that are ancestors of $z$ and descendents
  of nodes in $\mathbb{T}$.
  Initialize grad_table, a data structure associating tensors to their gradients
  grad_table[$z$] $\leftarrow 1$
  for $\mathbf{V}$ in $\mathbb{T}$ do
    build_grad($\mathbf{V}, \mathcal{G}, \mathcal{G}'$, grad_table)
  end for
  Return grad_table restricted to $\mathbb{T}$
```



Algorithm 6.6 The inner loop subroutine `build_grad`($\mathbf{V}, \mathcal{G}, \mathcal{G}'$, `grad_table`) of the back-propagation algorithm, called by the back-propagation algorithm defined in algorithm 6.5.

```
Require: $\mathbf{V}$, the variable whose gradient should be added to $\mathcal{G}$ and grad_table.
Require: $\mathcal{G}$, the graph to modify.
Require: $\mathcal{G}'$, the restriction of $\mathcal{G}$ to nodes that participate in the gradient.
Require: grad_table, a data structure mapping nodes to their gradients
  if $\mathbf{V}$ is in grad_table then
    Return grad_table[$\mathbf{V}$]
  end if
  $i \leftarrow 1$
  for $\mathbf{C}$ in get_consumers($\mathbf{V}, \mathcal{G}'$) do
    op $\leftarrow$ get_operation($\mathbf{C}$)
    $\mathbf{D} \leftarrow$ build_grad($\mathbf{C}, \mathcal{G}, \mathcal{G}'$, grad_table)
    $\mathbf{G}^{(i)} \leftarrow$ op.bprop(get_inputs($\mathbf{C}, \mathcal{G}'$), $\mathbf{V}, \mathbf{D}$)
    $i \leftarrow i + 1$
  end for
  $\mathbf{G} \leftarrow \sum_i \mathbf{G}^{(i)}$
  grad_table[$\mathbf{V}$] $= \mathbf{G}$
  Insert $\mathbf{G}$ and the operations creating it into $\mathcal{G}$
  Return $\mathbf{G}$
```

We can analyze computational cost of backprop through the # of operations executed (in this case, we're considering operations to have a nonvariable runtime, which is not actually true)
Computing a gradient in a graph with n nodes will never execute more than O(n^2) ops since at worst, the forward prop stage will execute all n nodes in the original graph

Backprop adds one Jacobian-vector product, O(1) nodes, per edge in the original graph
Since the computational edge is directed acyclic, it has at most O(n^2) edges

In pracice, most graphs are more efficient; most neural net cost functions are chain structed, causing backprop to have O(n) cost

This is better than the naive approach, which might execute exponentially many nodes

$$\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{\substack{\text{path}\,(u^{(\pi_1)},u^{(\pi_2)},\ldots,u^{(\pi_t)}),\\ \text{from}\,\pi_1=j\,\text{to}\,\pi_t=n}} \prod_{k=2}^{t} \frac{\partial u^{(\pi_k)}}{\partial u^{(\pi_{k-1})}}.$$

Non-recursive representation

Since the # of paths from node j to n can grow exponentially in the length of the path, the number of such paths (used above) grows exponentially w/ depth of forward prop

This is because $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ would be done so many times; to avoid recomputation, backprop uses dynamic programming which stores intermediate results in a table

Example on one layer MLP

Minibatch of examples as a design matrix X w/ a vector of corresponding class labels y

Network computes a layer of hidden features $\boldsymbol{H} = \max\{0, \boldsymbol{X}\boldsymbol{W}^{(1)}\}$ (we do not use bias)

We assume that our graph contains operations like ReLU and cross entropy

Predictions of unnormalized log probabilities over classes are given by $\boldsymbol{H}\boldsymbol{W}^{(2)}$.

The total cost, with regularization, will be

$$J = J_{\text{MLE}} + \lambda \left(\sum_{i,j} \left(W_{i,j}^{(1)} \right)^2 + \sum_{i,j} \left(W_{i,j}^{(2)} \right)^2 \right)$$

We wish to compute both $\nabla_{\boldsymbol{W}^{(1)}} J$ and $\nabla_{\boldsymbol{W}^{(2)}} J$; there are two paths leading backwards from J to the weights: one through cross entropy, and one through weight decay

Weight decay will always contribute 2(lambda)W^i to the gradient on W^i

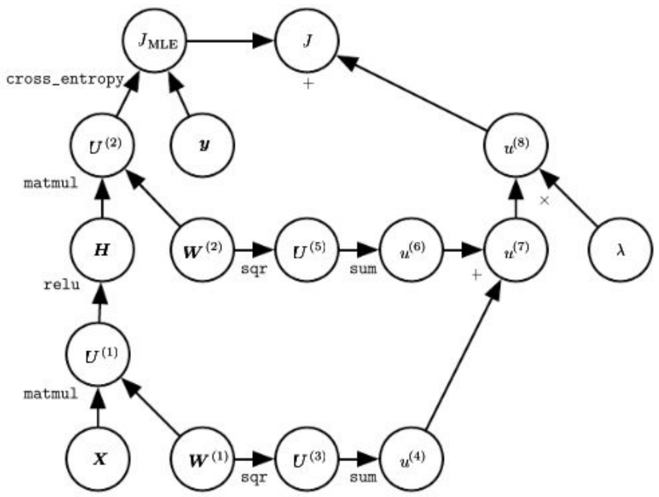


Figure 6.11: The computational graph used to compute the cost used to train our example of a single-layer MLP using the cross-entropy loss and weight decay.

Let G be the gradient on the unnormalized log probabilities U^2 provided by xentropy
Backprop needs to explore two branches: on the first, it adds $H^T G$ to the gradient on W^2 using backprop to derive the gradient of matrix multiplication
The other branch corresponds to the chain extending further along the network: the network computes $\nabla_{\boldsymbol{H}} J = \boldsymbol{G}\boldsymbol{W}^{(2)\top}$ for the first argument of matrix multiplication, and relu uses backprop to zero out components of the gradient for entries of U which were less than 0, which results in G'
Backprop rule for the second arg of matmul adds $W^T G$ to the gradient on W'

For an MLP, computational cost is dominated by matmul
During forward prop, multiply by each weight matrix resulting in O(w) multadds
During backprop, we multiply by the transpose of each weight matrix, which has the same cost
We need to store the input to the nonlinearity of the hidden layer, which is stored until backward pass has returned to the same point; the memory cost is $O(mn_h)$ where m is the number of examples in the minibatch and n_h is the number of hidden units

Most software implementations in practice need to support operations that output more than 1 tensor; to be computationally efficient, we implement these as 1 operation w/ 2 outputs
Backprop involves summation of many tensors together; the naive approach which computes each of these tensors separately can be avoided by adding them all onto a single buffer
Real world implementations of backprop must be specially designed based on data type (float)
Operations which have an undefined gradient may need to be altered for the user

Automatic differentiation - compute derivatives using an algorithm
Backprop is a subset of a group of techniques called reverse mode accumulation
Determining the optimal sequence of operations to compute the gradient is an NP problem

Suppose we have vars $p_1 \ldots p_n$ representing probabilities, and vars $z_1 .. z_n$ representing unnormalized log probabilities

$$q_i = \frac{\exp(z_i)}{\sum_i \exp(z_i)},$$

Softmax is built out of simple operations and our loss is $J = -\sum_i p_i \log q_i.$
It is easy to observe that dJ/dz is just $q_i - p_i$ but backprop cannot compute this and will instead use many nodes in the original graph

When the forward graph G has a single output node and each partial derivative $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ can be computed w/ a constant amount of computation, # of computations for gradient of backprop is the same as forward prop
The overall computation is O(#edges), but this can be reduced by simplifying the graph

Backprop can be extended to compute a Jacobian for k different scalar nodes or a tensor node containing k values
A naive implementation would perform k times more computation (k gradients instead of 1)
If the # of outputs > # of inputs, it is sometimes preferable to use forward mode accumulation
This avoids the need to store values and gradients for the graph, trading off for memory

Relationship between forward and backward is analogous to left-multiplying vs right-multiplying:
$ABCD$ where the matrices are Jacobian

- D is a column vector and A has many rows, this matrix product results in 1 output
- Multiplying by starting from the right requires matrix-vector products, which correpsonds to backwards mode
- Starting from the left involves matrix-matrix products, which makes the computation more expensive; however, if A has fewer rows than D has columns, it is cheaper to run left to right

Some software frameworks describe derivative expressions in the same way as the function being differentiated, which means symbolic differentiation can be applied to derivatives
We are usually interested in computing the Hessian: if f maps from R^n to R, then the Hessian is of size n x n (will double check this)
N is usually the # of parameters in the model, which could be in the billions, making the Hessian infeasible to represent

The alternative is to use Krylov methods, a set of iterative techniques for inverting a matrix or finding approximations to its eigenvectors / eigenvalues, using only matrix-vector products
To use these methods, we only need to be able to compute the product between a hessian matrix H and an arbitrary vector v, $\boldsymbol{H}\boldsymbol{v} = \nabla_{\boldsymbol{x}} \left[(\nabla_{\boldsymbol{x}} f(x))^\top \boldsymbol{v} \right]$
The outer gradient expression takes the gradient of a function of the inner gradient (obtained after multiplying by v)
If v is produced by the computational graph, autodiff should not differentiate through the graph which produced v
To compute Hessian itself, compute He^i where for i = {1, 2, …, n} where e^i_i is a one hot vector