Deep Learning

Muchang Bahng

Summer 2023

Contents

Unlike my machine learning notes, which focuses mostly on the theoretical soundness of classical (e.g. pre-deep and interpretable) models, the theory of deep neural networks have not been developed as well yet. Furthermore, the recency of developments, especially in the post-2010s, results in a pretty nonlinear[1] timeline that is difficult to categorize effectively. Before I give my motivation, the direct prerequisites for deep learning are basic knowledge of my notes in probability theory, machine learning, and real analysis.

Therefore, after much thought, I think organizing my notes in chronological order would be best. It turned out that in the early days of deep learning, most researchers like Andrew Ng stated that he focused on supervised models,[2] and it wasn't until the 2010s that the development of unsupervised models burgeoned.

1. We start by introducing *multilayered perceptrons* (MLPs), which build upon generalized linear models (GLMs) that we have went over in machine learning. This is pretty much the "foundational" model that we will build on. We then talk a about practical methodologies regarding training and control.
2. Then we introduce the other two architectures. The *convolutional neural network* (CNN) gives us a scalable way to perform sparse matrix multiplication efficiently, by taking advantage of locality. Then in *recurrent neural networks* (RNNs), we are not limited to a single n-dimensional vector input and are able to process a time series of inputs.
3. With these building blocks, we are able take and two neural networks together to create an encoder-decoder model. The two main applications of this was dimensionality reduction with *autoencoders*, followed by *seq2seq* for machine translation between sequences of words.
4. The practical success of autoencoders led to the development of not just density estimation models, but *generative* ones that seek to sample from the learned distribution. These generative models were "deep" extensions of the classical linear factor models resulting in *restricted Boltzmann machines (RBMs)* and *variational autoencoders (VAEs)*, which attempted to explicitly model an approximation of the true data generating distribution. More specifically, RBMs were *energy models* that borrowed ideas from physics to learn a distribution of the form $p_X(x) = \frac{1}{Z}e^{-f_\theta(x)}$. To sample from this, researchers already had access to the established Markov Chain Monte Carlo (MCMC) algorithms designed for this exact problem. As for VAEs, these were trained and sampled through *variational inference* methods.
5. In 2014, a completely new architecture composed of 2 neural networks competing each other gave rise to *generative adversarial networks* (GANs) which blew RBMs and VAEs away. A similar architecture followed with *generative stochastic networks* (GSNs).
6. In 2016, *normalizing flow models*, which attempted to model a smooth function that transformed simple latent variables to the true data generating distribution, were invented by Google DeepMind and surpassed GANs.
7. In 2017, *attention* took the world by storm, which had drastically improved machine translation in seq2seq, and the resulting *self-attention* plus the *transformer* architectures led to the success of OpenAI's ChatGPT.

Again, I emphasize that the math in these notes are not very advanced. However, implementing these simple models and training algorithms from scratch is a challenge in itself. I will go through implementing everything from scratch as if we were building a mini-version of PyTorch as we learn new topics. My implementations can be found here.

[1] No pun intended.
[2] He states this in his podcast with Lex Fridman.

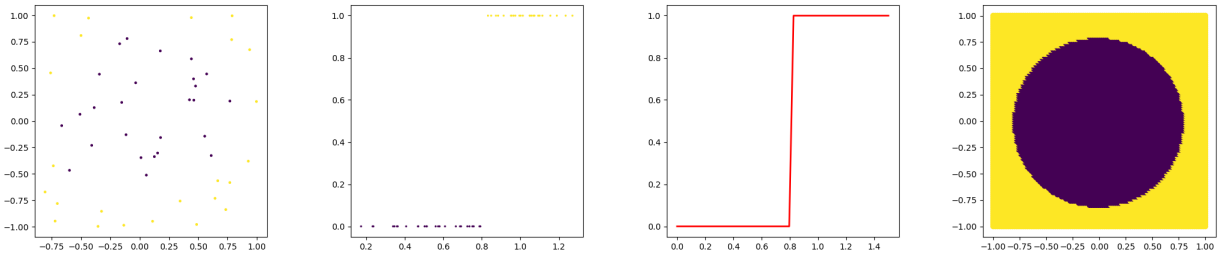
1 Multi-Layered Perceptrons

Why do we even *need* deep neural networks in the first place? Well with classical machine learning, the models that we worked with were "simple' enough[3] to interpret. For example, linear regression had a simple parametric form, and decision trees, though nonlinear and nonparametric, were very interpretable both in training and prediction. However, it may be that the very notion of interpretability is limiting the power of our models. Maybe the world is so complex that it may not be even possible to model it with these interpretable functions. Therefore, we should try to create a model with a simple enough parametrization, but something that is expressive enough to cover an extremely large space of functions. This may sound contradictory, but it is possible. We build upon what we already know: generalized linear models.

In simple regression, we transform the inputs into the relevant features $\mathbf{x}_n \mapsto \boldsymbol{\phi}(\mathbf{x}_n) = \boldsymbol{\phi}_n$ and then, when we construct a generalized linear model, we assume that the conditional distribution $Y \mid X = x$ is in the canonical exponential family, with some natural parameter $\eta(x)$ and expected mean $\mu(x) = \mathbb{E}[Y \mid X = x]$. Then, to choose the link function g that related $g(\mu(x)) = x^T\beta$, we can set it to be the canonical link g that maps μ to η. That is, we have $g(\mu(x)) = x^T\beta = \eta(x)$ such that the natural parameter is linearly dependent on the input. The inverse g^{-1} of the link function is called the **activation function**, which connects the expected mean to a linear function of x.

$$h_\beta(x) = g^{-1}(x^T\beta) = \mu(x) = \mathbb{E}[Y \mid X = x] \tag{1}$$

Now, note that for a classification problem, the decision boundary defined in the $\boldsymbol{\phi}$ feature space is linear, but it may not be linear in the input space $\mathcal{X}$.



(a) Data in space $\mathcal{X} = \mathbb{R}^2$. (b) Transformed data $\phi(\mathbf{x}) = \|\mathbf{x}\|$. (c) Logistic fit in transformed space. (d) Logistic fit to data in input space.

Figure 1: Consider the set of points in $\mathbb{R}^2$ with the corresponding class. We transform the features to $\boldsymbol{\phi}(x_1, x_2) = x_1^2 + x_2^2$, which gives us a new space to work with. Fitting logistic regression onto this gives a linear decision boundary in the space $\boldsymbol{\phi}$, but the boundary is circular in $\mathcal{X} = \mathbb{R}^2$.

We would like to extend this model by making the basis functions $\boldsymbol{\phi}_n$ depend on the parameters $\mathbf{w}$ and then allow these parameters to be adjusted during training.

1.1 Feedforward Fully-Connected Networks

So how should we construct parametric nonlinear basis functions? One way is to have a similar architecture as GLMs by having a linear map followed by an activation function $f(x) = \sigma(w^Tx + b)$. The simplest such function with the activation function as the step function

$$f(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases} \tag{2}$$

[3] I am using simple in a very loose sense, since this doesn't really mean that the model space is finite-dimensional or have any correlation with complexity.

is the perceptron algorithm. It divides $\mathbb{R}^d$ using a hyperplane $\boldsymbol{\omega}^T\mathbf{x} + b = 0$ and linearly classifies all points on one side to value 1 and the other side to value 0. This is similar to a neuron, which takes in a value and outputs a "signal" if the function evaluated gets past a threshold. However, for reasons regarding training these networks, we would like to use smooth activation functions for this, so we would use different activations. Hence we have a neuron.

Definition 1.1 (Neuron)

A **neuron** is a function of form

$$y = \sigma(\mathbf{w}^T x + b) \tag{3}$$

where $\sigma : \mathbb{R} \to \mathbb{R}$ is any nonlinear function, called an **activation functions**.

Ultimately, a neural net is really just a generalized linear model with some trained feature extractors, which is why in practice, if researchers want to predict a smaller dataset, they take a pretrained model on a related larger dataset and simply tune the final layer, since the second last layer most likely encodes all the relevant features. This is called *transfer learning*. But historically, it was called a *multilayer perceptron* and the name stuck.

Definition 1.2 (Feedforward, Fully-Connected Multilayer Perceptron)

A L-layer **multilayer perceptron (MLP)** $f_\theta : \mathbb{R}^D \to \mathbb{R}^M$, with parameters θ, is a function of form

$$h_\theta(\mathbf{x}) \coloneqq \sigma^{[L]} \circ W^{[L]} \circ \sigma^{[L-1]} \circ W^{[L-1]} \circ \cdots \circ \sigma^{[1]} \circ W^{[1]}(\mathbf{x}) \tag{4}$$

where $\sigma^{[l]} : \mathbb{R}^{N^{[l]}} \to \mathbb{R}^{N^{[l]}}$ is an activation function and $W^{[l]} : \mathbb{R}^{N^{[l-1]}} \to \mathbb{R}^{N^{[l]}}$ is an affine map. We will use the following notation.

1. The inputs will be labeled $\mathbf{x} = a^{[0]}$ which is in $\mathbb{R}^{N^{[0]}} = \mathbb{R}^D$.
2. We map $a^{[l]} \in \mathbb{R}^{N^{[l]}} \mapsto W^{[l+1]}a^{[l]} + b^{[l+1]} = z^{[l+1]} \in \mathbb{R}^{N^{[l+1]}}$, where z denotes a vector after an affine transformation.
3. We map $z^{[l+1]} \in \mathbb{R}^{N^{[l+1]}} \mapsto \sigma(z^{[l+1]}) = a^{[l+1]} \in \mathbb{R}^{N^{[l+1]}}$, where a denotes a vector after an activation function.
4. We keep doing this until we reach the second last layer with vector $a^{[L-1]}$.
5. Now we want our last layer to be our predicted output. Based on our assumptions of the problem, we construct a generalized linear model with some inverse link function g. We perform one more affine transformation $a^{[L-1]} \mapsto W^{[L]}a^{[L-1]} + b^{[L]} = z^{[L]}$, followed by the link function to get our prediction: $a^{[L]} = g(z^{[L]}) = h_\theta(\mathbf{x}) \in \mathbb{R}^M$.

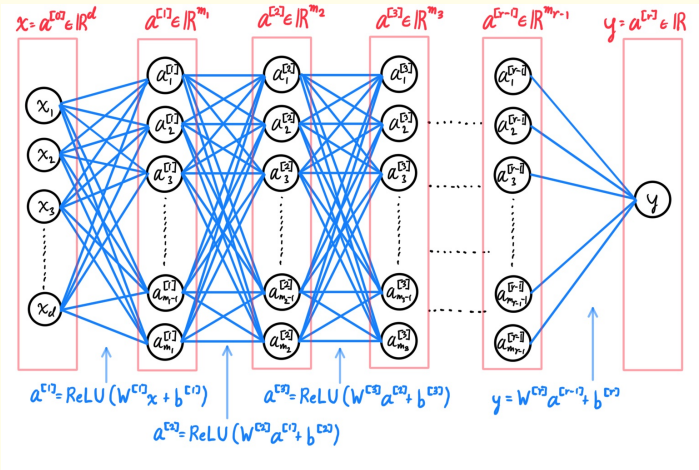


Figure 2: If there does not exist any edge from a potential input x to an output y, then this means that x is not relevant in calculating y. However, we usually work with **fully-connected neural networks**, which means that every input is relevant to calculating every output, since we usually cannot make assumptions about which variables are relevant or not. We can stack multiple neurons such that one neuron passes its output as input into the next neuron, resulting in a more complex function.

Note that each layer corresponds to how close a neuron is to the output. But really any neuron can be a function of any other neuron. For example, we can connect a neuron from layer 4 back to a neuron of layer 1. For now, we will consider networks that are restricted to a **feed-forward** architecture, in other words having no closed directed cycles.

Code 1.1 (Parameters and Neural Nets in PyTorch)

At this point, you have learned the theory of MLPs. To actually implement them in PyTorch, look at this module here, which will tell you on how to construct linear maps and activations functions, and more importantly see how you can look at the weights, modify them, and see how they are initialized. You can then learn how to explore the weights and biases of a neural network.

1.2 Function Space and Universal Approximation

Great, so we have defined our architecture, but how do we know that this class of functions is expressive? Neural networks have been mathematically studied back in the 1980s, and the reason that they are so powerful is that we can theoretically prove the limits on what they can learn. For very specific classes of functions, the results are easier, but for more general ones, it becomes much harder. We prove one of the theorems below.

Let us think about how one would construct approximations for such functions. Like in measure theory, we can think of every measurable function as a linear combination of a set of bump functions, and so we can get a neural network to do the same.

Example 1.1 (Bump Functions in $\mathbb{R}$)

Assuming the sigmoid activation function is used, the bump function

$$f(x) = \begin{cases} 1 & \text{if } a < x < b \\ 0 & \text{if else} \end{cases} \tag{5}$$

can be approximated by taking a linear combination of a sigmoid function stepping up and one

stepping down. That is,

$$f(x) \approx \frac{1}{2}\sigma\big(k(x-a)\big) - \frac{1}{2}\sigma\big(k(x-b)\big) \tag{6}$$

where k is a scaling constant that determines how steep the steps are for each function. Therefore, as $k \to \infty$, the function begins to look more like a step function.

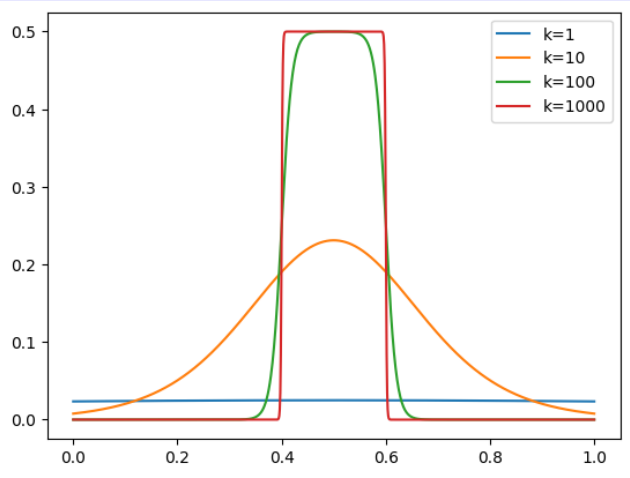


Figure 3: Bump function approximated with $a = 0.4, b = 0.6$, with differing values of k.

Example 1.2 (Bump Functions in $\mathbb{R}^2$)

To do this for a 2-D step function, of the form

$$f(x_1, x_2) = \begin{cases} 1 \text{ if } a < x_1 < b \\ 0 \qquad\qquad\quad \text{if else} \end{cases} \tag{7}$$

this is a simple extension of the first one. We just don't need to make our linear combination dependent on x_2 and we're done.

$$f(x) \approx \frac{1}{2}\sigma\big(k(x_1-a)\big) - \frac{1}{2}\sigma\big(k(x_1-b)\big) \tag{8}$$

Example 1.3 (Tower Functions in $\mathbb{R}^2$)

Now to construct a tower function of the form

$$f(x_1, x_2) = \begin{cases} 1 & \text{if } a_1 < x_1 < b_1, a_2 < x_2 < b_2 \\ 0 & \text{if else} \end{cases} \tag{9}$$

we need slightly more creativity. Now we can approximate it by doing

$$f(x) \approx \sigma\bigg(k_2\big[\sigma\big(k_1(x_1-a_1)\big) - \sigma\big(k_1(x_1-b_1)\big) + \sigma\big(k_1(x_2-a_2)\big) - \sigma\big(k_1(x_2-b_2)\big)big\big] - b_2\bigg) \tag{10}$$

At this point, we can see how this would extend to $\mathbb{R}^n$, and by isolating parts of the network we can have it approximate tower functions that are completely separate from each other, at any height, and then finally take a linear combination of them to approximate the original function of interest.

Theorem 1.1 (CS671 Fall 2023 PS5)

Suppose you have a 2D, L-lipschitz function $f(x_1, x_2)$ defined on a unit square ($x_1, x_2 \in [0, 1]$). You want to approximate this with an arbitrary neural net $\tilde{f}$ such that

$$\sup_{x \in [0,1]^2} |f(x) - \tilde{f}(x)| \leq \epsilon \tag{11}$$

If we divide the square into a checkerboard of $K \times K$ nonoverlapping squares, approximate the restriction of f to each subsquare with a tower function, what is the least K we would need to ensure that the error is less than ϵ?

Early in the development of the theory of neural nets, An Mei Chen, (currently VP of engineering in Qualcomm) showed that for certain neural networks, there are multiple parameters θ that map to the same function f. [3]

Theorem 1.2 (Parameter Symmetry)

Consider a 2-layer feedforward network of form

$$f = W^{[2]} \circ \sigma \circ W^{[1]} \tag{12}$$

where $\sigma = \tanh$. Let z be the hidden vector. We can see that by changing the signs of the ith row of $W^{[1]}$, z_i's sign will be flipped. From the properties that tanh is an odd function (i.e. $\tanh(-x) = -\tanh(x)$), therefore the activation will be also sign-flipped, but this effect can be negated by flipping the ith column of the $W^{[2]}$. Therefore, given that $z \in R^N$, i.e there are N hidden units, we can choose any set of row-column pairs of the weight matrices to invert, leading to a total of 2^N different weightings that produce the same function f.
Similarly, imagine that we permute the columns of $W^{[2]}$ and rows of $W^{[1]}$ in the same way. Then this will also lead to an invariance in f, and so this leads to $N!2^N$ different weight vectors that lead to the same function!

1.3 Tensors

In numerical computing packages like `numpy` in Python and `eigen` in C++, we often work with scalars, vectors, and matrices. From linear algebra, the generalization of these objects is a tensor, which is an element of a tensor product space.[4] The full mathematical abstraction is rarely needed in practice, and so developers call tensors by their realization as *multidimensional arrays*.

Definition 1.3 (Tensor)

A **tensor** is an element of a tensor product space $\bigotimes_i V_i$. It is represented as a **multidimensional array** of shape $(\dim(V_1), \dots, \dim(V_n))$.

If we were trying to build a `Tensor` class from scratch, what attributes should it have? Well obviously we need the actual data in the tensor, which we will call `storage`, plus some metadata about the `shape` (in math, known as the tensor *rank*). Usually, these packages optimize as much as possible for efficiency, and so these are implemented as C-style arrays, which then requires knowledge of the type of each element of the Tensor, called the `dtype`. Great, with these three attributes, we can do almost every type of arithmetic manipulation. Let's first introduce the most basic math tensor operations, which includes the normal operations supported in an algebra, plus some other ones. We will denote the shapes as well.

1. *Tensor Addition.*
2. *Tensor Additive Inverse.*

[4] For a refresher, look at my linear algebra notes.

3. *Scalar Multiplication.*
4. *Matrix Multiplication.*
5. *Elementwise Multiplication.*
6. *Elementwise Multiplicative Inverse.*
7. *Transpose.*

We would probably like some constructors that allows you to directly initialize tensors filled with 0s (`zeros`), 1s (`ones`), a multiplicative identity (`eye`[5]) Some random initializers would be good, such as sampling from uniforms (`uniform`), `gaussians` (`gaussian`, `randn`).

Finally, we would like some very fundamental operations, such as typecasting, comparison, and indexing as well.

[5]homophone for I, used to denoted the identity matrix.

Algorithm 1 Tensor Class Implementation

class Tensor:
Attributes:
  storage: array ▷Underlying data storage
  shape: tuple ▷Dimensions of tensor
  dtype: type ▷Data type of elements
Constructors:
`def __init__(data, shape, dtype):`
    Initialize tensor with given data, shape, and dtype
Static Constructors:
`@staticmethod`
`def zeros(shape, dtype):`
    return tensor filled with zeros
`@staticmethod`
`def ones(shape, dtype):`
    return tensor filled with ones
`@staticmethod`
`def eye(n, dtype):`
    return n×n identity matrix
`@staticmethod`
`def uniform(shape, low, high, dtype):`
    return tensor with uniform random values
`@staticmethod`
`def gaussian(shape, mean, std, dtype):`
    return tensor with gaussian random values
Arithmetic Operations:
`def __add__(self, other):`
    return element-wise addition
`def __neg__(self):`
    return additive inverse
`def __mul__(self, other):`
    return scalar or element-wise multiplication
`def matmul(self, other):`
    return matrix multiplication
`def __truediv__(self, other):`
    return element-wise division
`def transpose(self):`
    return transposed tensor
Utility Operations:
`def __repr__(self):`
    return string representation
`def __str__(self):`
    return human-readable string
`def __getitem__(self, index):`
    return indexed value(s)
`def __eq__(self, other):`
    return element-wise equality comparison

Note that there are other operations, such as concatenation, splitting, and stacking that would be a good idea to implement.

1.3.1 Strides

A specific property of PyTorch is that they use strides as another source of metadata in storing tensors, which greatly speeds up operations. Consider that we want to transpose the first two dimensions of a tensor. Then, we would have to create a new tensor and fill it in with all the elements, which may be too computationally expensive for such a small operation.

Definition 1.4 (Stride)

Given a tensor T of size $(N_1, \dots N_M)$, it is stored as a contiguous array of $\prod_m T_m$ elements, and we can index it as

$$T[n_1, n_2, \dots, n_M], \quad 1 \leq n_i \leq N_i \tag{13}$$

To counteract this, the **stride** is a array S of M elements,

$$S = (S_1, \dots, S_M) \tag{14}$$

where indexing with some $I = (n_1, \dots, n_M)$ is equivalent to computing $S \cdot I$ and taking that index in the array in memory. It defines a mapping.

If we do some calculation, the default stride of such a vector is defined

$$S_m = \prod_{m<j} N_j, \quad S_M = 1 \tag{15}$$

Example 1.4 (Transposing)

If we want to transpose the tensor above, then we change the stride from

$$S = \left(\prod_{1<j} N_j, \prod_{2<j} N_j, \dots, 1 \right) \tag{16}$$

to

$$S = \left(\prod_{2<j} N_j, \prod_{1<j} N_j, \dots, 1 \right) \tag{17}$$

1.3.2 Automatic Differentiation

Lemma 1.1 (Derivative of $+/-$)

Given two tensors X, Y and $Z_+ = X + Y, Z_- = X - Y$, we have

$$\frac{\partial Z_+}{\partial X} = +1 \qquad \frac{\partial Z_+}{\partial Y} = +1 \tag{18}$$

$$\frac{\partial Z_-}{\partial X} = +1 \qquad \frac{\partial Z_-}{\partial Y} = -1 \tag{19}$$

where ± 1 are tensors of 1 or -1s of the same shape as X, Y.

Lemma 1.2 (Derivative of Element-wise Multiplication)

Given two tensors X, Y and $Z = X \odot Y$, we have

$$\frac{\partial Z}{\partial X} = Y \qquad \frac{\partial Z}{\partial Y} = X \tag{20}$$

Lemma 1.3 (Derivative of Matrix Multiplication)

Given $X \in (N, M)$ and $Y \in (M, P)$, with $Z = XY \in (N, P)$, the derivative of matrix multiplication is

$$\frac{\partial Z}{\partial X} \in (N, P, N, M) \qquad \left(\frac{\partial Z}{\partial X}\right)_{i,j,k,l} := \frac{\partial Z_{i,j}}{\partial X_{k,l}} \tag{21}$$

$$\frac{\partial Z}{\partial Y} \in (N, P, M, P) \qquad \left(\frac{\partial Z}{\partial Y}\right)_{i,j,k,l} := \frac{\partial Z_{i,j}}{\partial Y_{k,l}} \tag{22}$$

1.4 Forward and Back Propagation

Back in the supervised learning notes, we have gone through the derivation for linear, logistic, and softmax regression. It turns out that despite them having very different architectures, with a identity, sigmoid, and softmax activation function, our choice of loss to be the mean squared loss, the binary cross-entropy, and the cross-entropy loss, had given very cute formulas in computing the gradient of the loss. Unfortunately, the formulas do not get cute when we differentiate neural networks, but they do come in a very structured way. To gain intuition, I would recommend to go over the exercises at the end of the chapter labeled ECE 689 Fall 2021 Midterm. If you just use chain rule to do the calculations, you can see that they require us to compute all the intermediate $z^{(i)}$'s and the $a^{(i)}$'s, a process called *forward propagation*, before we compute the gradients.

Definition 1.5 (Forward Propagation)

Given an MLP f and an input x, the process of sequentially evaluating

$$x = a^{[0]} \mapsto z^{[1]} \mapsto a^{[1]} \mapsto \ldots \mapsto z^{[L]} \mapsto a^{[L]} = f(x) \tag{23}$$

is called **forward propagation**.

When we want to compute the derivative of f. we can see that the intermediate partial derivatives in the chain rule are repeatedly used. That is, if we have layer $0 \leq l \leq L$, then to compute the derivative with respect to the lth layer we use the chain rule

$$\frac{\partial f}{\partial z^{[l]}} = \frac{\partial f}{\partial z^{[l+1]}} \cdot \frac{z^{[l+1]}}{z^{[l]}} \tag{24}$$

which requires us to know the derivative at the $(l+1)$th layer, along with the current values of $z^{[l]}, z^{[l+1]}$ to evaluate the derivatives at the current point. Therefore, we must complete forward propagation first and then compute *backwards* from the result to the input to compute the gradients.

Definition 1.6 (Backward Propagation)

Given an MLP f with input x that has been forward propagated, the process of sequentially evaluating

$$\frac{\partial f}{\partial a^{[L]}} \mapsto \frac{\partial f}{\partial z^{[L]}} \mapsto \ldots \mapsto \frac{\partial f}{\partial z^{[L]}}, \tag{25}$$

is called **backward propagation**, or **backprop**.

Backpropagation is not hard, but it is cumbersome notation-wise. What we really want to do is just compute a very long vector with all of its partials $\partial E / \partial \boldsymbol{\theta}$.

Algorithm 1.1 (Backpropagation)

To compute $\frac{\partial E_n}{\partial w_{ji}^{[l]}}$, it would be natural to split it up into a portion where E_n is affected by the term before activation $\mathbf{z}^{[l]}$ and how that is affected by $w_{ji}^{[l]}$. The same goes for the bias terms.

$$\frac{\partial E_n}{\partial w_{ji}^{[l]}} = \underbrace{\frac{\partial E_n}{\partial \mathbf{z}^{[l]}}}_{1\times N^{[l]}} \cdot \underbrace{\frac{\partial \mathbf{z}^{[l]}}{\partial w_{ji}^{[l]}}}_{N^{[l]}\times 1} \text{ and } \frac{\partial E_n}{\partial b_i^{[l]}} = \underbrace{\frac{\partial E_n}{\partial \mathbf{z}^{[l]}}}_{1\times N^{[l]}} \cdot \underbrace{\frac{\partial \mathbf{z}^{[l]}}{\partial b_i^{[l]}}}_{N^{[l]}\times 1} \tag{26}$$

It helps to visualize that we are focusing on

$$\mathbf{h}_{\boldsymbol{\theta}}(\mathbf{x}) = g\big(\dots \sigma(\underbrace{\mathbf{W}^{[l]}\mathbf{a}^{[l-1]} + \mathbf{b}^{[l]}}_{\mathbf{z}^{[l]}})\dots\big) \tag{27}$$

We can expand $\mathbf{z}^{[l]}$ to get

$$\mathbf{z}^{[l]} = \begin{pmatrix} w_{11}^{[l]} & \dots & w_{1N^{[l-1]}}^{[l]} \\ \vdots & \ddots & \vdots \\ w_{N^{[l]}1}^{[l]} & \dots & w_{N^{[l]}N^{[l-1]}}^{[l]} \end{pmatrix} \begin{pmatrix} a_1^{[l-1]} \\ \vdots \\ a_{N^{[l-1]}}^{[l-1]} \end{pmatrix} + \begin{pmatrix} b_1^{[l]} \\ \vdots \\ b_{N_{[l]}^{[l]}} \end{pmatrix} \tag{28}$$

$w_{ji}^{[l]}$ will only show up in the jth term of $\mathbf{z}^{[l]}$, and so the rest of the terms in $\frac{\partial \mathbf{z}^{[l]}}{\partial w_{ji}^{[l]}}$ will vanish. The same logic applies to $\frac{\partial \mathbf{z}^{[l]}}{\partial b_i^{[l]}}$, and so we really just have to compute

$$\frac{\partial E_n}{\partial w_{ji}^{[l]}} = \underbrace{\frac{\partial E_n}{\partial z_j^{[l]}}}_{1\times 1} \cdot \underbrace{\frac{\partial z_j^{[l]}}{\partial w_{ji}^{[l]}}}_{1\times 1} = \delta_j^{[l]} \cdot \frac{\partial z_j^{[l]}}{\partial w_{ji}^{[l]}} \text{ and } \frac{\partial E_n}{\partial b_i^{[l]}} = \underbrace{\frac{\partial E_n}{\partial z_j^{[l]}}}_{1\times 1} \cdot \underbrace{\frac{\partial z_j^{[l]}}{\partial b_i^{[l]}}}_{1\times 1} = \delta_j^{[l]} \cdot \frac{\partial z_j^{[l]}}{\partial b_i^{[l]}} \tag{29}$$

where the $\delta_j^{[l]}$ is called the jth **error term** of layer l. If we look at the evaluated jth row,

$$z_j^{[l]} = w_{j1}^{[l]} a_1^{[l-1]} + \dots w_{jN^{[l-1]}} a_{N^{[l-1]}}^{[l-1]} + b_j^{[l]} \tag{30}$$

We can clearly see that $\frac{\partial z_j^{[l]}}{\partial w_{ji}^{[l]}} = a_i^{[l-1]}$ and $\frac{\partial z_j^{[l]}}{\partial b_i^{[l]}} = 1$, which means that our derivatives are now reduced to

$$\frac{\partial E_n}{\partial w_{ji}^{[l]}} = \delta_j^{[l]} a_i^{[l-1]}, \qquad \frac{\partial E_n}{\partial b_i^{[l]}} = \delta_j^{[l]} \tag{31}$$

What this means is that we must know the intermediate values $\mathbf{a}^{[l-1]}$ beforehand, which is possible since we would compute them using forward propagation and store them in memory. Now note that the partial derivatives at this point have been calculated without any consideration of a particular error function or activation function. To calculate $\boldsymbol{\delta}^{[L]}$, we can simply use the chain rule to get

$$\delta_j^{[L]} = \frac{\partial E_n}{\partial z_j^{[L]}} = \frac{\partial E_n}{\partial \mathbf{a}^{[L]}} \cdot \frac{\partial \mathbf{a}^{[L]}}{\partial z_j^{[L]}} = \sum_k \frac{\partial E_n}{\partial a_k^{[L]}} \cdot \frac{\partial a_k^{[L]}}{\partial z_j^{[L]}} \tag{32}$$

which can be rewritten in the matrix notation

$$\boldsymbol{\delta}^{[L]} = \left(\frac{\partial \mathbf{g}}{\partial \mathbf{z}^{[L]}}\right)^T \left(\frac{\partial E_n}{\partial \mathbf{a}^{[L]}}\right) = \underbrace{\begin{bmatrix} \frac{\partial g_1}{\partial z_1^{[L]}} & \dots & \frac{\partial g_{N^{[L]}}}{\partial z_1^{[L]}} \\ \vdots & \ddots & \vdots \\ \frac{\partial g_1}{\partial z_{N^{[L]}}^{[L]}} & \dots & \frac{\partial g_{N^{[L]}}}{\partial z_{N^{[L]}}^{[L]}} \end{bmatrix}}_{N^{[L]}\times N^{[L]}} \begin{bmatrix} \frac{\partial E_n}{\partial a_1^{[L]}} \\ \vdots \\ \frac{\partial E_n}{\partial a_{N^{[L]}}^{[L]}} \end{bmatrix} \tag{33}$$

Note that as soon as we make a model assumption on the form of the conditional distribution $Y \mid X = x$ (e.g. it is Gaussian), with it being in the exponential family, we immediately get two things: the loss function E_n (e.g. sum of squares loss), and the canonical link function $\mathbf{g}$

1. If we assume that $Y \mid X = x$ is Gaussian in a regression (of scalar output) setting, then our canonical link would be $g(x) = x$, which gives the sum of squares loss function. Note that since the output is a real-valued scalar, $\mathbf{a}^{[L]}$ will be a scalar (i.e. the final layer is one node, $N^{[L]} = 1$).

$$E_n = \frac{1}{2}(y^{(n)} - a^{[L]})^2 \tag{34}$$

   To calculate $\boldsymbol{\delta}^{[L]}$, we can simply use the chain rule to get

$$\delta^{[L]} = \frac{\partial E_n}{\partial z^{[L]}} = \frac{\partial E_n}{\partial a^{[L]}} \cdot \frac{\partial a^{[L]}}{\partial z^{[L]}} = a^{[L]} - y^{(n)} \tag{35}$$

2. For classification (of M classes), we would use the softmax activation function (with its derivative next to it for convenience)

$$\mathbf{g}(\mathbf{z}) = \mathbf{g}\left(\begin{bmatrix} z_1 \\ \vdots \\ z_M \end{bmatrix} \right) = \begin{bmatrix} e^{z_1} / \sum_k e^{z_k} \\ \vdots \\ e^{z_M} / \sum_k e^{z_k} \end{bmatrix}, \quad \frac{\partial g_k}{\partial z_j} = \begin{cases} g_j(1 - g_j) & \text{if } k = j \\ -g_j g_k & \text{if } k \neq j \end{cases} \tag{36}$$

   which gives the cross entropy error

$$E_n = -\mathbf{y}^{(n)} \cdot \ln\left(\mathbf{h}_{\boldsymbol{\theta}}(\mathbf{x}^{(n)})\right) = -\sum_i y_i^{(n)} \ln(a_i^{[L]}) \tag{37}$$

   where the $\mathbf{y}$ has been one-hot encoded into a standard unit vector in $\mathbb{R}^M$. To calculate $\boldsymbol{\delta}^{[L]}$, we can again use the chain rule again

$$\begin{aligned}
\delta_j^{[L]} &= \sum_k \frac{\partial E_n}{\partial a_k^{[L]}} \cdot \frac{\partial a_k^{[L]}}{\partial z_j^{[L]}} && (38)\\
&= -\sum_k \frac{y_k^{(n)}}{a_k^{[L]}} \cdot \frac{\partial a_k^{[L]}}{\partial z_j^{[L]}} && (39)\\
&= \left(-\sum_{k \neq j} \frac{y_k^{(n)}}{a_k^{[L]}} \cdot \frac{\partial a_k^{[L]}}{\partial z_j^{[L]}} \right) - \frac{y_j^{(n)}}{a_j^{[L]}} \cdot \frac{a_j^{[L]}}{\partial z_j^{[L]}} && (40)\\
&= \left(-\sum_{k \neq j} \frac{y_k^{(n)}}{a_k^{[L]}} \cdot -a_k^{[L]} a_j^{[L]} \right) - \frac{y_j^{(n)}}{a_j^{[L]}} \cdot a_j^{[L]}(1 - a_j^{[L]}) && (41)\\
&= a_j^{[L]} \underbrace{\sum_k y_k^{(n)}}_{1} - y_j^{(n)} = a_j^{[L]} - y_j^{(n)} && (42)
\end{aligned}$$

   giving us

$$\boldsymbol{\delta}^{[L]} = \mathbf{a}_j^{[L]} - \mathbf{y}^{[L]} \tag{43}$$

Now that we have found the error for the last layer, we can continue for the hidden layers. We can again expand by chain rule that

$$\delta_j^{[l]} = \frac{\partial E_n}{\partial z_j^{[l]}} = \frac{\partial E_n}{\partial \mathbf{z}^{[l+1]}} \cdot \frac{\partial \mathbf{z}^{[l+1]}}{\partial z_j^{[l]}} = \sum_{k=1}^{N^{[l+1]}} \frac{\partial E_n}{\partial z_k^{[l+1]}} \cdot \frac{\partial z_k^{[l+1]}}{\partial z_j^{[l]}} = \sum_{k=1}^{N^{[l+1]}} \delta_k^{[l+1]} \cdot \frac{\partial z_k^{[l+1]}}{\partial z_j^{[l]}} \tag{44}$$

By going backwards from the last layer, we should already have the values of $\delta_k^{[l+1]}$, and to compute the second partial, we recall the way a was calculated

$$z_k^{[l+1]} = b_k^{[l+1]} + \sum_{j=1}^{N^{[l]}} w_{kj}^{[l+1]} \sigma(z_j^{[l]}) \implies \frac{\partial z_k^{[l+1]}}{\partial z_j^{[l]}} = w_{kj}^{[l+1]} \cdot \sigma'(z_j^{[l]}) \tag{45}$$

Now this is where the "back" in backpropagation comes from. Plugging this into the equation yields a final equation for the error term in hidden layers, called the **backpropagation formula**:

$$\delta_j^{[l]} = \sigma'(z_j^{[l]}) \sum_{k=1}^{N^{[l+1]}} \delta_k^{[l+1]} \cdot w_{kj}^{[l+1]} \tag{46}$$

which gives the matrix form

$$\boldsymbol{\delta}^{[l]} = \boldsymbol{\sigma}'(\mathbf{z}^{[l]}) \odot (\mathbf{W}^{[l+1]})^T \boldsymbol{\delta}^{[l+1]} = \begin{bmatrix} \sigma'(z_1^{[l]}) \\ \vdots \\ \sigma'(z_{N^{[L]}}^{[l]}) \end{bmatrix} \odot \begin{bmatrix} w_{11}^{[l+1]} & \dots & w_{N^{[l+1]}1}^{[l+1]} \\ \vdots & \ddots & \vdots \\ w_{1N^{[l]}}^{[l+1]} & \dots & w_{N^{[l+1]}N^{[l]}}^{[l+1]} \end{bmatrix} \begin{bmatrix} \delta_1^{[l+1]} \\ \vdots \\ \delta_{N^{[l+1]}}^{[l+1]} \end{bmatrix} \tag{47}$$

and putting it all together, the partial derivative of the error function E_n with respect to the weight in the hidden layers for $1 \leq l < L$ is

$$\frac{\partial E_n}{\partial w_{ji}^{[l]}} = a_i^{[l-1]} \sigma'(z_j^{[l]}) \sum_{k=1}^{N^{[l+1]}} \delta_k^{[l+1]} \cdot w_{kj}^{[l+1]} \tag{48}$$

A little fact is that the time complexity of both forward prop and back prop should be the same, so if you ever notice that the time to compute these two functions scales differently, you're probably making some repeated calculations somewhere.

Algorithm 1.2 (Epoch of Training)

Before training, we initialize all the parameters to be

$$\boldsymbol{\theta} = (\mathbf{W}^{[1]}, \mathbf{b}^{[1]}, \mathbf{W}^{[2]}, \dots, \mathbf{W}^{[L]}, \mathbf{b}^{[L]}) \tag{49}$$

Then, we repeat the following for one epoch of training.

1. *Choose Batch*: We choose an arbitrary data point $(\mathbf{x}^{(n)}, \mathbf{y}^{(n)})$, an minibatch, or the entire batch to compute the gradients on.
2. *Forward Propagation*: Apply input vector $\mathbf{x}^{(n)}$ and use forward propagation to compute the values of all the hidden and activation units

$$\mathbf{a}^{[0]} = \mathbf{x}^{(n)}, \mathbf{z}^{[1]}, \mathbf{a}^{[1]}, \dots, \mathbf{z}^{[L]}, \mathbf{a}^{[L]} = h_{\boldsymbol{\theta}}(\mathbf{x}^{(n)}) \tag{50}$$

3. *Back Propagation*:
   (a) Evaluate the $\boldsymbol{\delta}^{[l]}$'s starting from the back with the formula

$$\boldsymbol{\delta}^{[L]} = \left(\frac{\partial \mathbf{g}}{\partial \mathbf{z}^{[L]}} \right)^T \left(\frac{\partial E_n}{\partial \mathbf{a}^{[L]}} \right) \tag{51}$$

$$\boldsymbol{\delta}^{[l]} = \boldsymbol{\sigma}'(\mathbf{z}^{[l]}) \odot (\mathbf{W}^{[l+1]})^T \boldsymbol{\delta}^{[l+1]} \quad l = 1, \dots, L-1 \tag{52}$$

where $\frac{\partial \mathbf{g}}{\partial \mathbf{z}^{[L]}}$ can be found by taking the derivative of the known link function, and the rest of the terms are found by forward propagation (these are all functions which have been fixed in value by inputting $\mathbf{x}^{(n)}$).

(b) Calculate the derivatives of the error as

$$\frac{\partial E_n}{\partial \mathbf{W}^{[l]}} = \boldsymbol{\delta}^{[l]}(\mathbf{a}^{[l-1]})^T, \qquad \frac{\partial E_n}{\partial \mathbf{b}^{[l]}} = \boldsymbol{\delta}^{[l]} \tag{53}$$

4. *Gradient Descent*: Subtract the derivatives with step size α. That is, for $l = 1, \dots, L$,

$$\mathbf{W}^{[l]} = \mathbf{W}^{[l]} - \alpha \frac{\partial E_n}{\partial \mathbf{W}^{[l]}}, \qquad \mathbf{b}^{[l]} = \mathbf{b}^{[l]} - \alpha \frac{\partial E_n}{\partial \mathbf{b}^{[l]}} \tag{54}$$

The specific optimizer can differ, e.g. Adam, SGD, BFGS, etc., but the specific algorithm won't be covered here. It is common to use Adam, since it usually works better. If we can afford to iterate over the entire batch, L-BFGS may also be useful.

Code 1.2 (Neural Net from Scratch)

Now it's time to implement what most newcomers fear most: a neural net from scratch using only numpy. Doing this will get you to understand the inner workings of a neural net, and you can find the relevant code here.

Code 1.3 (Pytorch Implementation of Forward and Backward Propagation)

Once you have finished implementing from scratch, you can now use the PyTorch API to access the same model weights. The code here shows how to look at the forward propagation and backpropagation steps in PyTorch in intermediate layers and shows the backend behind storing gradients.

1.5 Overparamterization is Good?

Given that the input dimension is D, say that all the hidden layers are of dimension D and we have L layers. Then, we are storing a matrix (plus bias vector) at each layer, resulting in a scaling of $O(D^2L)$. This quadratic scaling leads to overparameterized models, which should raise a red flag. This naturally leads to overfitting, but a strange phenomenon occurs.[6]

1. In the beginning, the training loss goes down along with the validation loss.
2. Soon the validation loss starts to go up while the training loss goes down, leading to overfitting.
3. The overfitting is worst when the training loss is 0.
4. At this point, the training loss remains at 0, but generalization starts to improve, and mysteriously the validation loss starts going down.

There are many theories of why the last step ever happens. To interpret this, let's revisit what overfitting means. It means that small perturbations of the inputs will result in large variances in the outputs. If we generalize well, $x + \epsilon$ should also result in $f(x) + O(\epsilon)$. Therefore, this means that the more parameters it has, the better this stability is and therefore the more robust the model. How should we measure this sense of stability? In analysis, a metric to assess the robustness of a deep neural net $f_\theta : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ is its Lipshitz constant, which effectively bounds how much f can change given some change in x.

Definition 1.7 (Lipshitz Continuity)

A function $f : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ is called **Lipshitz continuous** if there exists a constant L such that for all $x, y \in \mathbb{R}^n$

$$||f(x) - f(y)||_2 \leq L||x - y||_2 \tag{55}$$

[6] I found this from Lex Fridman's podcast with Ilya Sutskever.

and the smallest L for which the inequality is true is called the **Lipshitz constant**, denoted $\text{Lip}(f)$.

Theorem 1.3 (Lipschitz Upper Bound with Operator Norm of Total Derivative)

If $f : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ is Lipschitz continuous, then

$$\text{Lip}(f) = \sup_{x \in \mathbb{R}^n} ||D_x f||_{\text{op}} \tag{56}$$

where $||\cdot||_{\text{op}}$ is the operator norm of a matrix. In particular, if f is scalar-valued, then its Lipschitz constant is the maximum norm of its gradient on its domain

$$\text{Lip}(f) = \sup_{x \in \mathbb{R}^n} ||\nabla f(x)||_2 \tag{57}$$

The above theorem makes sense, since indeed the stability of the function should be equal to the stability of its "maximum" linear approximation $D_x f$.

Theorem 1.4 (Lipschitz Upper Bound for MLPs)

It has already been shown that for a K-layer MLP

$$h_\theta(\mathbf{x}) \coloneqq \mathbf{T}_K \circ \boldsymbol{\rho}_{K-1} \circ \mathbf{T}_{K-1} \circ \cdots \circ \boldsymbol{\rho}_1 \circ \mathbf{T}_1(\mathbf{x}) \tag{58}$$

the Lipshitz constant for an affine map $\mathbf{T}_k(\mathbf{x}) = M_k \mathbf{x} + b_k$ is simply the operator norm (largest singular value) of M_k, while that of an activation function is always bounded by some well-known constant, usually 1. So, the Lipshitz constant of the entire composition h is simply the product of all operator norms of M_k.

What about K-computable functions in general? That is, given a function $f : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ with

$$\begin{aligned}
v_0(\mathbf{x}) &= \mathbf{x} && (59)\\
v_1(\mathbf{x}) &= g_1\big(v_0(\mathbf{x})\big) && (60)\\
v_2(\mathbf{x}) &= g_2\big(v_0(\mathbf{x}), v_1(\mathbf{x})\big) && (61)\\
&\dots = \dots && (62)\\
v_k(\mathbf{x}) &= g_k\big(v_0(\mathbf{x}), v_1(\mathbf{x}), \dots, v_{k-1}(\mathbf{x})\big) && (63)\\
&\dots = \dots && (64)\\
v_K(\mathbf{x}) &= g_K\big(v_0(\mathbf{x}), v_1(\mathbf{x}), \dots, v_{K-2}(\mathbf{x}), v_{K-1}(\mathbf{x})\big) && (65)
\end{aligned}$$

where $v_k : \mathbb{R}^n \longrightarrow \mathbb{R}^{n_k}$, with $n_0 = n$ and $n_K = m$, and

$$g_k : \prod_{i=0}^{k-1} \mathbb{R}^{n_i} \longrightarrow \mathbb{R}^{n_k} \tag{66}$$

To differentiate v_k w.r.t. $\mathbf{x}$, we can use the chain rule, resulting in the total derivative

$$\underbrace{\frac{\partial v_k}{\partial \mathbf{x}}}_{n_k \times n} = \sum_{i=1}^{k-1} \underbrace{\frac{\partial g_k}{\partial v_i}}_{n_k \times n_i} \underbrace{\frac{\partial v_i}{\partial \mathbf{x}}}_{n_i \times n} \tag{67}$$

Therefore, it is this Lipschitz property that might entail how stable an MLP is. We have seen from the universal approximation theorems that for a data set of any size, we can always fit a one-layer perceptron

that perfectly fits through all of them, given that the layer is large enough. In these cases, we are interested in fitting the data *smoothly*, and theoretical research in bounding the Lipschitz constant is popular.

In practice this behavior is reflected in *most* cases, but they may be very sensitive in some cases, which we call *adversarial examples*. Adversarial examples take advantage of this weakness by adding carefully chosen perturbations that drastically change the output of the network. Adversarial machine learning attempts to study these weaknesses and hopefully use them to create more robust models. It is natural to expect that the precise configuration of the minimal necessary perturbations is a random artifact of the normal variability that arises in different runs of backpropagation learning. Yet, it has been found that adversarial examples are relatively robust, and are shared by neural networks with varied number of layers, activations or trained on different subsets of the training data. This suggest that the deep neural networks that are learned by backpropagation have *intrinsic* blind spots, whose structure is connected to the data distribution in a non-obvious way.

1.6 Engineering Heuristics

1.7 Optimizers

We have assumed knowledge of gradient descent in the back propagation step in the previous section, but let's revisit this by looking at linear regression. Given our dataset $\mathcal{D} = \{\mathbf{x}^(n), y^{(n)}\}$, we are fitting a linear model of the form

$$f(\mathbf{x}; \mathbf{w}, b) = \mathbf{w}^T \mathbf{x} + b \tag{68}$$

The squared loss function is

$$\mathcal{L}(\mathbf{w}, b) = \frac{1}{2} \sum_{n=1}^{N} \big(y - f(\mathbf{x}; \mathbf{w}, b)\big)^2 = \frac{1}{2} \sum_{n=1}^{N} \big(y - (\mathbf{w}^T \mathbf{x} + b)\big)^2 \tag{69}$$

If we want to minimize this function, we can visualize it as a d-dimensional surface that we have to traverse. Recall from multivariate calculus that the gradient of an arbitrary function $\mathcal{L}$ points in the steepest direction in which $\mathcal{L}$ increases. Therefore, if we can compute the gradient of $\mathcal{L}$ and step in the *opposite direction*, then we would make the more efficient progress towards minimizing this function (at least locally). The gradient can be solved using chain rule. Let us solve it with respect to $\mathbf{w}$ and b separately first. Beginners might find it simpler to compute the gradient element-wise.

$$\begin{aligned}
\frac{\partial}{\partial w_j} \mathcal{L}(\mathbf{w}, b) &= \frac{\partial}{\partial w_j} \left(\frac{1}{2} \sum_{n=1}^{N} \Big(f(\mathbf{x}^{(n)}; \mathbf{w}, b) - y^{(n)} \Big)^2 \right) && (70) \\
&= \frac{1}{2} \sum_{n=1}^{N} \frac{\partial}{\partial w_j} \Big(f(\mathbf{x}^{(n)}; \mathbf{w}, b) - y^{(n)} \Big)^2 && (71) \\
&= \frac{1}{2} \sum_{n=1}^{N} 2 \Big(f(\mathbf{x}^{(n)}) - y^{(n)} \Big) \cdot \frac{\partial}{\partial w_j} \big(f(\mathbf{x}^{(n)}; \mathbf{w}, b) - y^{(n)} \big) && (72) \\
&= \frac{1}{2} \sum_{n=1}^{N} 2 \Big(f(\mathbf{x}^{(n)}) - y^{(n)} \Big) \cdot \frac{\partial}{\partial w_j} \big(\mathbf{w}^T \mathbf{x}^{(n)} + b - y^{(n)} \big) && (73) \\
&= \sum_{n=1}^{N} \big(f(\mathbf{x}^{(n)}; \mathbf{w}, b) - y^{(n)} \big) \cdot x_j^{(n)} \qquad (\text{for } j = 0, 1, \ldots, d) && (74)
\end{aligned}$$

As for getting the derivative w.r.t. b, we can redo the computation and get

$$\frac{\partial}{\partial w_j} \mathcal{L}(\mathbf{w}, b) = \sum_{n=1}^{N} \big(f(\mathbf{x}^{(n)}; \mathbf{w}, b) - y^{(n)} \big) \tag{75}$$

and in the vector form, setting $\boldsymbol{\theta} = (\mathbf{w}, b)$, we can set

$$\nabla \mathcal{L}(\mathbf{w}) = \mathbf{X}^T(\hat{\mathbf{y}} - \mathbf{y}) \tag{76}$$
$$\nabla \mathcal{L}(b) = (\hat{\mathbf{y}} - \mathbf{y}) \cdot \mathbf{1} \tag{77}$$

where $\hat{\mathbf{y}}_n = f(\mathbf{x}^{(n)}; \mathbf{w}, b)$ are the predictions under our current linear model and $\mathbf{X} \in \mathbb{R}^{n \times d}$ is our design matrix. This can easily be done on a computer using a package like `numpy`. Remember that GD is really just an algorithm that updates $\boldsymbol{\theta}$ repeatedly until convergence, but there are a few problems.

1. The algorithm can be susceptible to local minima. A few countermeasures include shuffling the training set or randomly choosing initial points θ
2. The algorithm may not converge if α (the step size) is too high, since it may overshoot. This can be solved by reducing the α with each step, using *schedulers*.
3. The entire training set may be too big, and it may therefore be computationally expensive to update $\boldsymbol{\theta}$ as a whole, especially if $d >> 1$. This can be solved using stochastic gradient descent.

Rather than updating the vector $\boldsymbol{\theta}$ in batches, we can apply **stochastic gradient descent** that works incrementally by updating $\boldsymbol{\theta}$ with each term in the summation. That is, rather than updating as a batch by performing the entire matrix computation by multiplying over N dimensions,

$$\nabla \mathcal{L}(\mathbf{w}) = \underbrace{\mathbf{X}^T}_{D \times N} \underbrace{(\hat{\mathbf{y}} - \mathbf{y})}_{N \times 1} \tag{78}$$

we can reduce this load by choosing a smaller subset $\mathcal{M} \subset \mathcal{D}$ of $M < N$ elements, which gives

$$\nabla \mathcal{L}_{\mathcal{M}}(\mathbf{w}) = \underbrace{\mathbf{X}_{\mathcal{M}}^T}_{D \times M} \underbrace{(\hat{\mathbf{y}_{\mathcal{M}}} - \mathbf{y})}_{\mathcal{M}})_{M \times 1} \tag{79}$$

The reason we can do this is because of the following fact.

Theorem 1.5 (Unbiasedness of SGD)

$\nabla \mathcal{L}_{\mathcal{M}}(\mathbf{w})$ is an *unbiased estimator* of the true gradient. That is, setting $\mathcal{M}$ as a random variable of samples over $\mathcal{D}$, we have

$$\mathbb{E}_{\mathcal{M}}[\nabla \mathcal{L}_{\mathcal{M}}(\mathbf{w})] = \nabla \mathcal{L}(\mathbf{w}) \tag{80}$$

Proof.

We use linearity of expectation for all $\mathcal{M} \subset \mathcal{D}$ of size M.

Even though these estimators are noisy, we get to do much more iterations and therefore have a faster net rate of convergence. By using repeated chain rule, or a fancier term is automatic differentiation, as shown before, SGD can be used to optimize neural networks.

Extending beyond SGD, there are other optimizers we can use. Essentially, we are doing a highly nonconvex optimization, which doesn't have a straightforward answer, so the best we can do is play around with some properties. 0th order approximations are hopeless since the dimensions are too high, and second order approximations are hopeless either since computing the Hessian is too expensive for one run. Therefore, we must resort to some first order methods, which utilize the gradient. Some other properties to consider are:

1. Learning rate
2. Momentum
3. Batch Size

Now we list some of the most common optimizers and will compare their performance.

Definition 1.8 ()

Stochastic Gradient Descent TBD

Definition 1.9 ()

Adam TBD

Definition 1.10 ()

RMSProp TBD

Definition 1.11 ()

Adagrad TBD

Definition 1.12 ()

Nesterov Momentum TBD

Definition 1.13 ()

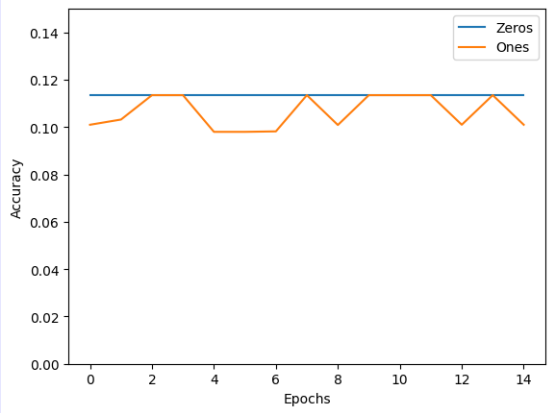
L-BFGS TBD

1.8 Weight Initialization

The way that we initialize our weights can have a huge impact on our training performance. Imagine that you are creating the first neural network and you want to decide how to initialize it. You may consider many different cases.

Example 1.5 (Constant Initialization)

You may first think of initializing everything to 0 or 1, which is the simplest. Let's run this, but we can already see by epoch 15 that we have some problems.



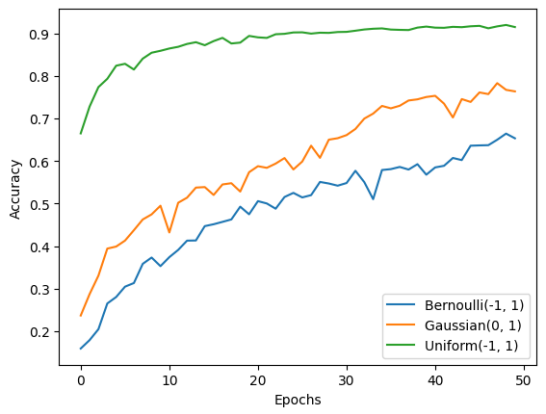
Clearly, this is not good, and theoretically this makes sense since it means all our activations are going to be the same, and thus all our gradients will be the same, meaning that are updates will be the same for every weight, which is not good mixing. We can see this below:

Example 1.6 (Random Initialization with High Variance)

Okay, this didn't work, so perhaps you think it would be a good idea have more randomness to the initialization so that all the weights aren't exactly one number. You could think of initializing everything with three distinct schemes:

1. Randomly initialize everything to be -1 or 1 with equal probability.
2. Randomly initialize everything to be a Gaussian random variable with standard deviation 1.
3. Randomly initialize everything to be a uniform random variable between -1 and 1.

Running the experiments give the following.

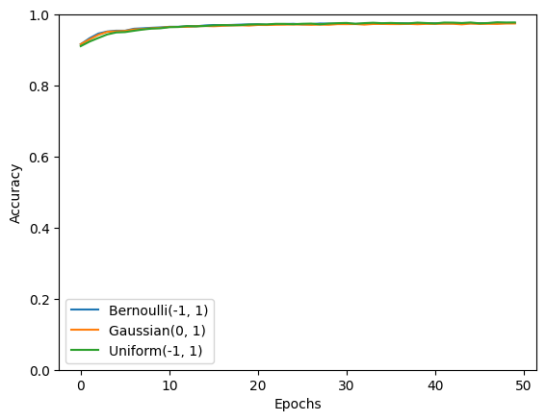


However, this is also not good since it means that the activations will be very large, and thus the gradients will be very large, and so the updates will be very large. This is not good since it means that the weights will be jumping around a lot, and we won't be able to converge. Furthermore, depending on what activations we choose, e.g. tanh or sigmoid, very large activations may saturate the gradients and kill the learning.

Example 1.7 (Random Initialization with Low Variance)

This improves the next problem but now you want to fix the situation of the gradients being too big. Therefore, you should initialize the parameters to be smaller values, but not so small that they are zeros and we have the same problem as before. We use improved schemes:

1. Randomly initialize everything to be -0.1 or 0.1 with equal probability.
2. Randomly initialize everything to be a Gaussian random variable with standard deviation 0.1.
3. Randomly initialize everything to be a uniform random variable between -0.1 and 0.1.



Through out experiments, we have learned that a good rule of thumb for initializing weights is to make them small and uniformly random without being too small. While it is harder to get better than this for MNIST, a slightly better approach is Xavier initialization, which builds upon our same ideas.

Definition 1.14 (Xavier Initialization)

The **Xavier initialization** simply initializes each weight as a uniform distribution, with its range dependent on the size of the input.

$$w_{ij}^{[l]} \sim U\left(-\frac{1}{\sqrt{N^{[l-1]}}}, \frac{1}{\sqrt{N^{[l-1]}}} \right) \tag{81}$$

where $N^{[l-1]}$ is the number of neurons in the previous layer. This is a good rule of thumb for the weights, but the biases can be initialized to 0 (though they are also initialized uniformly by default).

Code 1.4 (Experimenting with Weight Initializations)

The code used for generating the figures can be found here.

1.9 Activation Functions

The choice of the activation function can have a significant impact on your training, and we will describe a few examples below. The first thing to note is that we must ensure that there is a nonzero gradient almost everywhere. If, for example, we had a piecewise constant activation function, the gradient is 0 almost everywhere, and it would kill the gradient of the entire network. In the early days of deep learning, researchers used the probability-inspired sigmoid and tanh functions as the main source of nonlinearity. Let's go over them below.

Definition 1.15 (Sigmoid)

Sigmoid activations are historically popular since they have a nice interpretation as a saturating "fire rate" of a neuron. However, there are 3 problems:

1. The saturated neurons "kill" the gradients, since if the input at any one point in the layers is too positive or negative, the gradient will vanish, making very small updates. This is known as the **vanishing gradient problem**. Therefore, the more layers a neural network has, the more likely we are to see this vanishing gradient problem.
2. Sigmoid functions are not zero centered (i.e. its graph doesn't cross the point $(0, 0)$). Consider what happens when the input x to a neuron is always positive. Then, the sigmoid f will have a gradient of
$$f\left(\sum_i w_i x_i + b \right) \implies \frac{\partial f}{\partial w_i} = f'\left(\sum_i w_i x_i + b \right) x_i \tag{82}$$
which means that the gradients $\nabla_{\mathbf{w}} f$ will always have all positive elements or all negative elements, meaning that we will be restricted to moving in certain nonoptimal directions when updating our parameters.

Definition 1.16 (Hyperbolic Tangent)

The hyperbolic tangent is zero centered, which is nice, but it still squashes numbers to range $[-1, 1]$ and therefore kills the gradients when saturated.

It turns out that these two activations were ineffective in deep learning due to saturation. A less probability inspired activation was the ReLU, which showed better generalization an speed of convergence.

Definition 1.17 (Rectified Linear Unit)

The ReLU function has the following properties:

1. It does not saturate in the positive region.
2. It is very computationally efficient (and the fact that it is nondifferentiable at one point doesn't really affect computations).
3. It converges much faster than sigmoid/tanh in practice.
4. However, note that if the input is less than 0, then the gradient of the ReLU is 0. Therefore, if we input a vector that happens to have all negative values, then the gradient would vanish and we wouldn't make any updates. These ReLU "dead zones" can be a problem since it will never activate and never update, which can happen if we have bad initialization. A more common case is when your learning rate is too high, and the weights will jump off the data manifold.

Unfortunately, the ReLU had some weaknesses, mainly being the *dying ReLU*, which is when the ReLU is stuck in the negative region and never activates. This is a problem since the gradient is 0 in the negative region, and so the weights will never update. Therefore, some researchers have proposed some modifications to the ReLU.

Definition 1.18 (Leaky ReLU)

The leaky ReLU

$$\sigma(x) = \max\{0.01x, x\} \tag{83}$$

does not saturate (i.e. gradient will not die), is computationally efficient, and converges much faster than sigmoid/tanh in practice. We can also parameterize it with α and have the neural net optimize α along with the weights.

$$\sigma(x) = \max\{\alpha x, x\} \tag{84}$$

Definition 1.19 (ELU)

The exponential linear unit has all the benefits of ReLU, with closer to mean outputs. It has a negative saturation regime compared with leaky ReLU, but it adds some robustness to noise.

$$\sigma(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\big(\exp x - 1\big) & \text{if } x \leq 0 \end{cases} \tag{85}$$

Definition 1.20 (SELU)

The scaled exponential linear unit is a self-normalizing activation function, which means that it preserves the mean and variance of the input. This is useful for deep networks, since the mean and variance of the input will be preserved through the layers. Its formula is

$$\sigma(x) = \lambda \begin{cases} x & \text{if } x > 0 \\ \alpha\big(\exp x - 1\big) & \text{if } x \leq 0 \end{cases} \tag{86}$$

where λ and α are constants.

Later on, some further modifications were made, such as the **Swish** and the **Mish** [7] activation functions. These functions have a distinctive negative concavity, unlike ReLU, which accounts for preservation of small negative weights.

Definition 1.21 (Swish)

The Swish activation function is defined as

$$\sigma(x) = x \cdot \sigma(\beta x) \tag{87}$$

where β is a parameter that can be learned.

Definition 1.22 (Mish)

The Mish activation function is defined as

$$\sigma(x) = x \cdot \tanh(\ln(1 + \exp(x))) \tag{88}$$

Code 1.5 (Generating Graphs)

Code used to generate these graphs are here.

1.10 Datasets and Dataloaders

For here, we will go over some of the main datasets that are used in deep learning.

Definition 1.23 (MNIST and Fashion MNIST)

The MNIST dataset consists of 60k training images and 10k test images of handwritten digits. The Fashion MNIST dataset consists of 60k training images and 10k test images of clothing items. These are considered quite easy with the basic benchmarks:

1. Linear classifiers can reach past 90% accuracy.
2. A 2 layer MLP can reach up to 97% accuracy.
3. A CNN can reach up to 99% accuracy.

Definition 1.24 (CIFAR10 and CIFAR 100)

The CIFAR10 dataset consists of 60k 32x32 color images in 10 classes, with 6k images per class. The CIFAR100 dataset consists of 60k 32x32 color images in 100 classes, with 600 images per class. These are considered quite hard with the basic benchmarks:

1. Linear classifiers can reach past 40% accuracy.
2. A 2 layer MLP can reach up to 60% accuracy.
3. A CNN can reach up to 80% accuracy.

Definition 1.25 (ImageNet)

The ImageNet dataset, created at Stanford by Fei-Fei Li [4], consists of 1.2 million training images and 50k validation images in 1000 classes. This is considered very hard with the basic benchmarks.

Creating your own custom dataset with spreadsheets or images is easy.[7] Loading it to a dataloader that shuffles and outputs minibatches of data is trivial. However, when doing so, you should pay attention to a couple things.

1. Batch size: The dataloader stores the dataset (which can be several hundred GBs) in the drive, and extracts batches into memory for processing. You should set your batch sizes so that they can fit into the GPU memory, which is often smaller than the CPU memory.

[7] https://pytorch.org/tutorials/beginner/data_loading_tutorial.html

1.11 Exercises

Exercise 1.1 (Tarokh, ECE685 2021 Midterm 1.1)

Let $x \in \mathbb{R}$ denote a random variable with the following *cumulative distribution function*

$$F(x) = \exp\bigg(-\exp\bigg(-\frac{x-\mu}{\beta}\bigg)\bigg) \tag{89}$$

where μ and $\beta > 0$ denote the location and scale parameters, respectively. Let $\mathcal{D} = \{x_1, \dots, x_n\}$ be a set of n iid observations of x.

1. Write an equation for a cost function $L(\mu, \beta \mid \mathcal{D})$ whose minimization gives the maximum likelihood estimates for μ and β.
2. Compute the derivatives of $L(\mu, \beta \mid \mathcal{D})$ with respect to μ and β and write a system of equations whose solution gives the MLEs of μ and β.

Solution 1.1

We can derive the PDF of the observation as

$$f(x; \mu, \beta) = \frac{dF(x)}{dx} = \frac{1}{\beta}\exp\bigg\{ -\bigg(\frac{x-\mu}{\beta} + \exp\Big(-\frac{x-\mu}{\beta}\Big)\bigg)\bigg\} \tag{90}$$

and the likelihood is then

$$L(\mu, \beta \mid \mathcal{D}) = \prod_{i=1}^{N} \frac{1}{\beta}\exp\bigg\{ -\bigg(\frac{x^{(i)}-\mu}{\beta} + \exp\Big(-\frac{x^{(i)}-\mu}{\beta}\Big)\bigg)\bigg\} \tag{91}$$

Rather than maximizing this likelihood, we minimize the negative log of it, defined as

$$\ell(\mu, \beta \mid \mathcal{D}) = -\ln L(\mu, \beta \mid \mathcal{D}) = N\ln\beta + \frac{\sum_i x^{(I)} - N\mu}{\beta} + \sum_{i=1}^{N} \exp\Big(-\frac{x^{(i)}-\mu}{\beta}\Big) \tag{92}$$

The derivatives of ℓ can be computed simply by using the derivative rules.

$$\frac{\partial \ell}{\partial \mu} = -\frac{N}{\beta} + \frac{1}{\beta}\sum_{i=1}^{N} \exp\Big(-\frac{x^{(i)}-\mu}{\beta}\Big) \tag{93}$$

$$\frac{\partial \ell}{\partial \beta} = \frac{N}{\beta} - \frac{\sum_i x^{(i)} - N\mu}{\beta^2} + \frac{1}{\beta^2}\sum_{i=1}^{N}(x^{(i)}-\mu)\exp\Big(-\frac{x^{(i)}-\mu}{\beta}\Big) \tag{94}$$

and so the MLE estimates that minimizes ℓ can be found by setting the equations above equal to 0.

Exercise 1.2 (ECE 685 Fall 2021 Midterm 1.2)

The figure depicts a simple neural network with one hidden layer. The inputs to the network are denoted by x_1, x_2, x_3, and the output is denoted by y. The activation functions of the neurons in the hidden layer are given by $h_1(z) = \sigma(z), h_2(z) = \tanh(z)$, and the output unit activation function is $g(z) = z$, where $\sigma(z) = \frac{1}{1+\exp(-z)}$ and $\tanh(z) = \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)}$ are the logistic sigmoid and hyperbolic tangent, respectively. The biases b_1, b_2 are added to the inputs of the neurons int he hidden layer before passing them through the activation functions. let

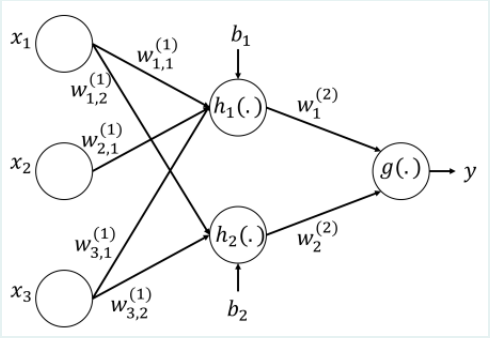
$$\mathbf{w} = (b_1, b_2, w_{11}^{(1)}, w_{12}^{(1)}, w_{21}^{(1)}, w_{31}^{(1)}, w_{32}^{(1)}, w_1^{(2)}, w_2^{(2)}) \tag{95}$$

denote the vector of network parameters.

1. Write the input output relation $y = f(x_1, x_2, x_3; \mathbf{w})$ in explicit form.
2. Let $\mathcal{D} = \{(x_{1,n}, x_{2,n}, x_{3,n})\}$ denote a training dataset of N points where $y_n \in \mathbb{R}$ are labels of the corresponding data points. We want to estimate the network parameters $\mathbf{w}$ using $\mathcal{D}$ by minimizing the mean squared error loss

$$L(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\big(f(x_{1,n}, x_{2,n}, x_{3,n}; \mathbf{w}) - y_n\big)^2 \tag{96}$$

   Compute the gradient of $L(\mathbf{w})$ with respect to the network parameters $\mathbf{w}$.
3. Write pseudo code for one iteration for minimizing $L(\mathbf{w})$ with respect to the network parameters $\mathbf{w}$ using SGD with learning rate $\eta > 0$.



Solution 1.2

We can write the computation graph as

$$z_1^{(1)} = w_{11}^{(1)}x_1 + w_{21}^{(1)}x_2 + w_{31}^{(1)}x_3 + b_1 \tag{97}$$
$$z_2^{(1)} = w_{12}^{(1)}x_1 + w_{32}^{(1)}x_3 + b_2 \tag{98}$$
$$a_1^{(1)} = \sigma(z^{(1)}) \tag{99}$$
$$a_2^{(1)} = \tanh(z_2^{(1)}) \tag{100}$$
$$z^{(2)} = w_1^{(2)}a_1^{(1)} + w_2^{(2)}a_2^{(1)} \tag{101}$$
$$y = a^{(2)} = g(z^{(2)}) \tag{102}$$

and composing these gives

$$y = w_1^{(2)}\sigma(w_{11}^{(1)}x_1 + w_{21}^{(1)}x_2 + w_{31}^{(1)}x_3 + b_1) + w_2^{(2)}\tanh(w_{12}^{(1)}x_1 + w_{32}^{(1)}x_3 + b_2) \tag{103}$$

The gradient of the network can be written as

$$\nabla_{\mathbf{w}}L(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\nabla_{\mathbf{w}}\big(f(x_{1,n}, x_{2,n}, x_{3,n}; \mathbf{w}) - y_n\big)^2 \tag{104}$$
$$= \sum_{n=1}^{N}(f(x_{1,n}, x_{2,n}, x_{3,n}; \mathbf{w}) - y_n)\nabla_{\mathbf{w}}f(x_{1,n}, x_{2,n}, x_{3,n}) \tag{105}$$

where

$$\nabla_{\mathbf{w}}f(x_{1,n}, x_{2,n}, x_{3,n}) = \left.\frac{\partial f}{\partial \mathbf{w}}\right|_{\mathbf{x}=\mathbf{x}^{(n)}} \tag{106}$$

Now we can take derivatives using chain rule, working backwards, and using the derivative identities

$\sigma'(z) = \sigma(z)(1-\sigma(z))$ and $\tanh'(z) = 1 - \tanh^2(z)$.

$$\frac{\partial f}{\partial w_1^{(2)}} = \frac{\partial f}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial w_1^{(2)}} = a_1^{(1)} \tag{107}$$

$$\frac{\partial f}{\partial w_2^{(2)}} = \frac{\partial f}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial w_2^{(2)}} = a_2^{(1)} \tag{108}$$

$$\frac{\partial f}{\partial w_{11}^{(1)}} = \frac{\partial f}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial a_1^{(1)}} \frac{\partial a_1^{(1)}}{\partial z_1^{(1)}} \frac{\partial z_1^{(1)}}{\partial w_{11}^{(1)}} = w_1^{(2)} a_1^{(1)} (1 - a_1^{(1)}) x_1 \tag{109}$$

$$\frac{\partial f}{\partial w_{21}^{(1)}} = \frac{\partial f}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial a_1^{(1)}} \frac{\partial a_1^{(1)}}{\partial z_1^{(1)}} \frac{\partial z_1^{(1)}}{\partial w_{21}^{(1)}} = w_1^{(2)} a_1^{(1)} (1 - a_1^{(1)}) x_2 \tag{110}$$

$$\frac{\partial f}{\partial w_{31}^{(1)}} = \frac{\partial f}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial a_1^{(1)}} \frac{\partial a_1^{(1)}}{\partial z_1^{(1)}} \frac{\partial z_1^{(1)}}{\partial w_{31}^{(1)}} = w_1^{(2)} a_1^{(1)} (1 - a_1^{(1)}) x_3 \tag{111}$$

$$\frac{\partial f}{\partial b_1} = \frac{\partial f}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial a_1^{(1)}} \frac{\partial a_1^{(1)}}{\partial z_1^{(1)}} \frac{\partial z_1^{(1)}}{\partial b_1} = w_1^{(2)} a_1^{(1)} (1 - a_1^{(1)}) \tag{112}$$

$$\frac{\partial f}{\partial w_{12}^{(1)}} = \frac{\partial f}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial a_2^{(1)}} \frac{\partial a_2^{(1)}}{\partial z_2^{(1)}} \frac{\partial z_2^{(1)}}{\partial w_{12}^{(1)}} = w_2^{(2)} (1 - (a_2^{(1)})^2) x_1 \tag{113}$$

$$\frac{\partial f}{\partial w_{13}^{(1)}} = \frac{\partial f}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial a_2^{(1)}} \frac{\partial a_2^{(1)}}{\partial z_2^{(1)}} \frac{\partial z_2^{(1)}}{\partial w_{13}^{(1)}} = w_2^{(2)} (1 - (a_2^{(1)})^2) x_3 \tag{114}$$

$$\frac{\partial f}{\partial b_2} = \frac{\partial f}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial a_2^{(1)}} \frac{\partial a_2^{(1)}}{\partial z_2^{(1)}} \frac{\partial z_2^{(1)}}{\partial b_2} = w_2^{(2)} (1 - (a_2^{(1)})^2) \tag{115}$$

To compute one step of SGD, we must first choose a minibatch $\mathcal{M} \subset \mathcal{D}$ and then compute

$$\nabla_{\mathbf{w};\mathcal{M}} L(\mathbf{w}) = \sum_{(\mathbf{x},y)\in\mathcal{M}} (f(\mathbf{x};\mathbf{w}) - y) \nabla_{\mathbf{w}} f(\mathbf{x}) \tag{116}$$

where we compute the gradient simply over the minibatch. Then, we update the parameters according to

$$\mathbf{w} = \mathbf{w} - \eta \nabla_{\mathbf{w};\mathcal{M}} L(\mathbf{w}) \tag{117}$$

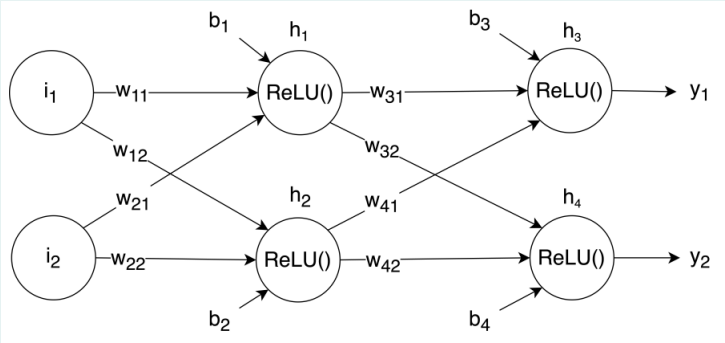
Exercise 1.3 (ECE 685 Fall 2021 Midterm 1.3)

Given the following neural network with 2 inputs (x_1, x_2), fully-connected layers and ReLU activations. The weights and biases of hidden units are denoted w and b, with h as activation units. For example,

$$h_1 = \text{ReLU}(x_1 w_{11} + x_2 w_{21} + b_1) \tag{118}$$

The outputs are denoted as (y_1, y_2) and the ground truth targets are denoted as (t_1, t_2).

$$y_1 = \text{ReLU}(h_1 w_{31} + h_2 w_{41} + b_3) \tag{119}$$



The values of the variables are given as follows:

i_1	i_2	w_{11}	w_{12}	w_{21}	w_{22}	w_{31}	w_{32}	w_{41}	w_{42}	b_1	b_2	b_3	b_4	t_1	t_2
1	2	1	0.5	-0.5	1	0.5	-2	-1	0.5	-0.5	-0.5	1	1	2	4

1. Compute the output (y_1, y_2) of the input (x_1, x_2) using the network parameters as specified above.
2. Compute the mean squared error of the computed output and the target labels.
3. Using the calculated MSE, update the weight w_{31} using GD with $\eta = 0.01$.
4. Do the same with weight w_{42}.
5. Do the same with weight w_{22}.

2 Training and Control

2.1 Early Stopping

Since neural networks are overparameterized, it makes sense that given enough training time, they will overfit to the training set. Therefore, you must stop training when the validation loss starts to decrease. This simple method is known as **early stopping**.

2.2 L1 and L2 Regularization

Another way to regularize is by simply adding in a L1 or L2 regularization term.

Sometimes, it may not always be the best idea to regularize a neural net equally through all weights. For example, weights which may be deeper down the forward pass may focus on more high level features and therefore should be regularized differently than those that are close to the input. Other types of regularization, such as Fiedler regularization [10] focuses on preserving the graph structure of the weights.

2.3 Dropout

Overfitting is always a problem. With unlimited computation, the best way to regularize a fixed-sized mdoel is to average the predictions of all possible settings of the parameters, weighting each setting by its posterior probability given the training the data. However, this is computationally expensive and cannot be done for moderately complex models.

The dropout method introduced by [9], addresses this issue. We literally drop out some features (not the weights!) before feeding them to the next layer by setting some activation functions to 0. Given a neural net of N total nodes, we can think of the set of its 2^N thinned subnetworks. For each training minibatch, a new thinned network is sampled and trained.

At each layer, recall that forward prop is basically

$$\begin{aligned}\mathbf{z}^{[l+1]} &= \mathbf{W}^{[l+1]}\mathbf{a}^{[l]} + \mathbf{b}^{[l+1]} \\ \mathbf{a}^{[l+1]} &= \boldsymbol{\sigma}(\mathbf{z}^{[l+1]})\end{aligned}$$

Now what we do with dropout is

$$\begin{aligned}r_j^{[l]} &\sim \text{Bernoulli}(p) \\ \tilde{\mathbf{a}}^{[l]} &= \mathbf{r}^{[l]} \odot \mathbf{a}^{[l]} \\ \mathbf{z}^{[l+1]} &= \mathbf{W}^{[l+1]}\tilde{\mathbf{a}}^{[l]} + \mathbf{b}^{[l+1]} \\ \mathbf{a}^{[l+1]} &= \boldsymbol{\sigma}(\mathbf{z}^{[l+1]})\end{aligned}$$

Basically we a sample a vector of 0s and 1s from a multivariate Bernoulli distribtion. We element-wise multiply it with $\mathbf{a}^{[l]}$ to create the thinned output $\tilde{\mathbf{a}}^{[l]}$. In test time, we do not want the stochasticity of having to set some activation functions to 0. That is, consider the neuron $\mathbf{a}^{[l]}$ and the random variable $\tilde{\mathbf{a}}^{[l]}$. The expected value of $\mathbf{z}^{[l+1]}$ is

$$\mathbb{E}[\mathbf{z}^{[l+1]}] = \mathbb{E}[\mathbf{W}^{[l+1]}\tilde{\mathbf{a}}^{[l]} + \mathbf{b}^{[l+1]}] = \mathbb{E}[\mathbf{W}^{[l+1]}\tilde{\mathbf{a}}^{[l]}] = p\mathbb{E}[\mathbf{W}^{[l+1]}\mathbf{a}^{[l]}]$$

and to make sure that the output at test time is the same as the expected output at training time, we want to multiply the weights by p: $W_{\text{test}}^{[l]} = p\, W_{\text{train}}^{[l]}$. Another way is to use **inverted dropout**, where we can divide by p in the training stage and keep the testing method the same.

Code 2.1 ()

The code here shows how to implement dropout in PyTorch, which uses dropout layers.

2.4 Data Augmentation

It is well known that having more training data helps with overfitting, and so we may be able to perform basic transformations to our current data to artificially generate more training data. For example, if we have images, then we can flip, crop, translate, rotate, stretch, shear, and lens-distort these images with the same label.

2.5 Normalization Layers

Just like how we have to normalize our data before we input into a linear model, it may help to normalize the outputs of one layer of a neural net before we input it into the next layer. This is an engineer's method to help with the training process. There are two ways that we can generally normalize data. First is to normalize each sample, known as **layer normalization**, and the other way is to normalize the samples over the batch.

Definition 2.1 (Layer Norm)

Given some batched output data $X \in \mathbb{R}^{b \times \mathbf{d}}$, where b represents the batch size and $\mathbf{d} = d_1 \times \ldots \times d_k$ the dimension of each sample, we can normalize each $x_i = X_{i,:}$ in the batch with **layer normalization** by

$$x_i \mapsto \frac{x_i - \mathbb{E}[x_i]}{\sqrt{\text{Var}[x_i] + \varepsilon}} \odot \gamma + \beta \tag{120}$$

where γ, β are learnable parameters that are the same shape as x_i. If X is of dimension $b \times \mathbf{d}$, we must use `nn.LayerNorm(d)` since these are the sizes of the learnable parameters.

Example 2.1 (Layer Norm)

The following example shows that each row (sample in batch) is normalized independently from one another.

```
ln = nn.LayerNorm(5)
x = torch.Tensor(range(10)).reshape(2, 5)
print(x)
tensor([[0., 1., 2., 3., 4.],
        [5., 6., 7., 8., 9.]])

print(ln(x))
tensor([[-1.4142, -0.7071,  0.0000,  0.7071,  1.4142],
        [-1.4142, -0.7071,  0.0000,  0.7071,  1.4142]],
       grad_fn=<NativeLayerNormBackward0>)
```

This also works for higher dimensions.

```
ln = nn.LayerNorm((5, 2))
x = torch.Tensor(range(20)).reshape(2, 5, 2)
print(x)
tensor([[[ 0.,  1.],
         [ 2.,  3.],
         [ 4.,  5.],
         [ 6.,  7.],
         [ 8.,  9.]],

        [[10., 11.],
         [12., 13.],
         [14., 15.],
```

```
          [16., 17.],
          [18., 19.]]])
print(ln(x))
tensor([[[-1.5667, -1.2185],
          [-0.8704, -0.5222],
          [-0.1741,  0.1741],
          [ 0.5222,  0.8704],
          [ 1.2185,  1.5667]],

         [[-1.5667, -1.2185],
          [-0.8704, -0.5222],
          [-0.1741,  0.1741],
          [ 0.5222,  0.8704],
          [ 1.2185,  1.5667]]], grad_fn=<NativeLayerNormBackward0>)
```

The tunable parameters γ, β are indeed the same size. They are initialized to 1s and 0s.

```
>>> for k, v in ln.state_dict().items():
...     print(k, v)
...
weight tensor([[1., 1.],
        [1., 1.],
        [1., 1.],
        [1., 1.]])
bias tensor([[0., 0.],
        [0., 0.],
        [0., 0.],
        [0., 0.]])
```

Definition 2.2 (Batch Norm)

Batch normalization targets each feature over all batches rather than each sample (like columns vs rows). Therefore, given some batched output data $X \in \mathbb{R}^{b \times \mathbf{d}}$, where b represents the batch size and $\mathbf{d} = d_1 \times \ldots \times d_k$ the dimension of each output, we can normalize each feature $x_i = X_{:, i \in \mathbf{d}}$ by

$$x_i \mapsto \frac{x_i - \mathbb{E}[x_i]}{\sqrt{\text{Var}[x_i] + \varepsilon}} \odot \gamma + \beta \tag{121}$$

where $\gamma, \beta \in \mathbb{R}^b$ are learnable parameters that are the same size as the batch. There are two types of batch norms implemented in pytorch.

1. If X has hyperdimension 2 with $b \times d$, we use `BatchNorm1d(d)` since we are normalizing over the batch for each feature and we have d features to normalize.
2. If X has hyperdimension 3 with $b \times d_1 \times d_2$, we use `BatchNorm1d(d_1)`.
3. If X has hyperdimension 4 with $b \times d_1 \times d_2 \times d_3$, we use `BatchNorm2d(d_1)`.

Example 2.2 (Batch Norm 1D)

We can see that each feature is normalized independently from one another. For 2D,

```
>>> bn = nn.BatchNorm1d(5)
>>> x = torch.Tensor(range(10)).reshape(2, 5)
```

```
>>> print(x)
tensor([[0., 1., 2., 3., 4.],
        [5., 6., 7., 8., 9.]])
>>> print(bn(x))
tensor([[-1.0000, -1.0000, -1.0000, -1.0000, -1.0000],
        [ 1.0000,  1.0000,  1.0000,  1.0000,  1.0000]],
       grad_fn=<NativeBatchNormBackward0>)
```

For 3D inputs,

```
>>> bn = nn.BatchNorm1d(5)
>>> x = torch.Tensor(range(30)).reshape(2, 5, 3)
>>> print(x)
tensor([[[ 0.,  1.,  2.],
         [ 3.,  4.,  5.],
         [ 6.,  7.,  8.],
         [ 9., 10., 11.],
         [12., 13., 14.]],

        [[15., 16., 17.],
         [18., 19., 20.],
         [21., 22., 23.],
         [24., 25., 26.],
         [27., 28., 29.]]])
>>> print(bn(x))
tensor([[[-1.1267, -0.9941, -0.8616],
         [-1.1267, -0.9941, -0.8616],
         [-1.1267, -0.9941, -0.8616],
         [-1.1267, -0.9941, -0.8616],
         [-1.1267, -0.9941, -0.8616]],

        [[ 0.8616,  0.9941,  1.1267],
         [ 0.8616,  0.9941,  1.1267],
         [ 0.8616,  0.9941,  1.1267],
         [ 0.8616,  0.9941,  1.1267],
         [ 0.8616,  0.9941,  1.1267]]], grad_fn=<NativeBatchNo
rmBackward0>)
```

Example 2.3 (Batch Norm 2D)

Here is an example of batch norm 2d. There really isn't a difference between these two methods except the dimension that they take in. That is all.

```
>>> bn = nn.BatchNorm2d(5)
>>> x = torch.Tensor(range(60)).reshape(2, 5, 3, 2)
>>> print(x)
tensor([[[[ 0.,  1.],
          [ 2.,  3.],
          [ 4.,  5.]],
          ...
          [58., 59.]]]])
>>> print(bn(x))
tensor([[[[-1.1592, -1.0929],
          [-1.0267, -0.9605],
          ...
```

```
            [ 1.0929,  1.1592]]]], grad_fn=<NativeBatchNormBack
ward0>)
```

2.6 Residual Connections

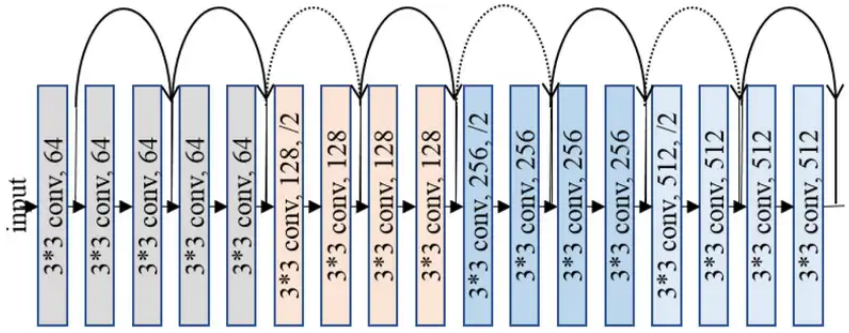


Figure 4: Resnet architecture.

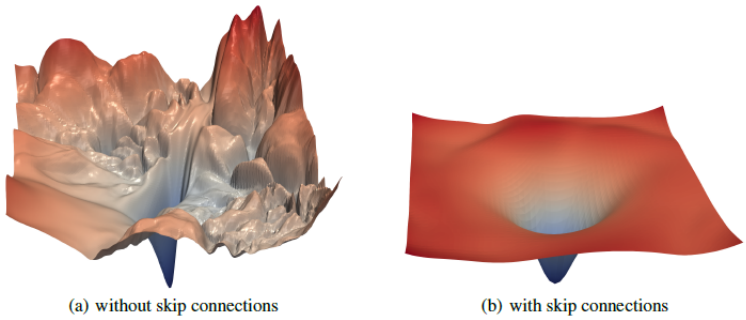


Figure 5: Low-dimensional visual of loss with vs without residual connections.

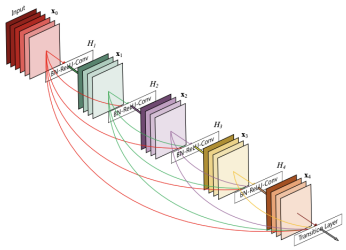
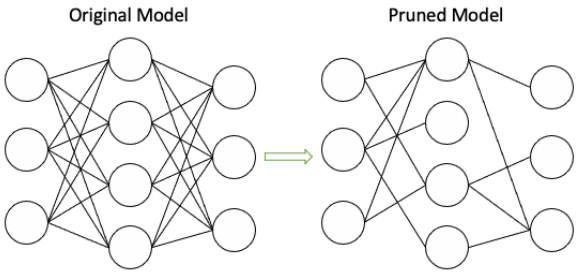


Figure 6: Densenet architecture.

2.7 Network Pruning

It can be computationally and memory intensive to train and utilize neural networks. This is where network pruning comes in, which attempts to identify a subnetwork that performs as well as the original. Given a neural net $f(\mathbf{x}, \boldsymbol{\theta})$ where $\boldsymbol{\theta} \in \mathbb{R}^M$, a pruned neural network can be thought of as a subnetwork $f(\mathbf{x}, \mathbf{m} \odot \boldsymbol{\theta})$, where $\mathbf{m}$ is a **mask**, i.e. a vector in $\{0, 1\}^M$ that, when multiplied component-wise to $\boldsymbol{\theta}$, essentially "deletes" a portion of the parameters.



This idea has been around for a long time, and the general method of pruning is as such:

1. We initialize the neural network $f(\mathbf{x}, \boldsymbol{\theta}_0)$ and train it until we have $f(\mathbf{x}, \boldsymbol{\theta})$.
2. We now prune the network. The most basic pruning scheme is to keep the top $k\%$ largest weights, since smaller weights do not contribute much to the forward prop, and thus can be ignored.

These pruned networks have been shown to reach accuracies as high as the original network, with equal training progress. Now, if we were to take only this pruned network and train it from the beginning, it will perform as well as the original network, *only under* the condition that we start from the same initialization $\mathbf{m} \odot \boldsymbol{\theta}$. If we take this subnetwork and initialize it differently at $\boldsymbol{\theta}_0'$, then this subnetwork would not train well. Therefore, the performance of the pruned network is dependent on the initialization!

If we had initialized the full network differently, trained it, and then pruned again, we may have a different subnetwork that will only train well on its own given this new initialization. Therefore, a good initialization is extremely important for training subnetworks. This fact doesn't help much since we can't just take some arbitrary subnetwork and train it since we don't know the good initialization. We must always train the full network, then find the subnetwork, and then find its initialization.

This is essentially the **lottery ticket hypothesis** [5], which states that a randomly-initialized, dense neural network contains a subnetwork that is initialized such that, when trained in isolation, it can match the test accuracy of the original network after training for at must the same number of iterations.

This paper hints at why neural networks work at all. It first states that only a very small subnetwork is responsible for the vast majority of its performance, but it must be initialized at the right position. But by overparameterizing these neural nets so much (by a certain margin), they have so many different combinations of subnetworks such that whatever initialization you throw at it, it is guaranteed that some subnetwork within it will train well with this initialization. This subnetwork is called the *winning ticket*.

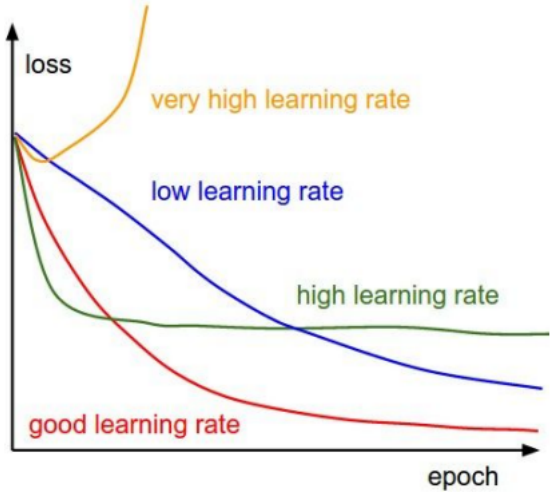
2.8 Summary

Here is a few steps you can take as a guide to training a neural network.

1. Preprocess the data.
2. Choose your neural net architecture (number of layers/neurons, etc.)
3. Do a forward pass with the initial parameters, which should be small, and check that the loss is reasonable (e.g. $\log(1/10) \approx 2.3$ for softmax classification of 10 classes).
4. Now crank up the regularization term, and your loss should have gone up.
5. Now try to train on only a very small portion of your data without regularization using SGD, which you should be able to overfit and get the accuracy to 100%.
6. Now you can train your whole dataset. Start off with a small regularization (e.g. 1e-6) and find a learning rate that makes the loss go down.
   (a) Run for a few epochs to see if the cost goes down too slowly (step size is too small) or the cost explodes (step size too big). A general tip is that if the cost is ever bigger than 3 times the original cost, then this is an indication that the cost has exploded.

(b) We can run a grid search (in log space) over the learning rate and the regularization hyperparameters over say 10 epochs each, and compare which one makes the most progress.

7. Monitor and visualize the loss curve.



If you see loss curves that are flat for a while and then start decreasing, then bad initialization is a prime suspect.

8. We also want to track the ratio of weight updates and weight magnitudes. That is, we can take the norm of the weights $\boldsymbol{\theta}$ and the gradient updates $\nabla\boldsymbol{\theta}$, and a rule of thumb is that the ratio should be about

$$\frac{||\nabla\boldsymbol{\theta}||}{||\boldsymbol{\theta}||} \approx 0.001 \text{ or } 0.01$$

3 Convolutional Neural Networks

Convolutional networks work with images, so let's introduce a nice way to represent them as vectors.

Definition 3.1 (Image)

An image is a vector in some tensor product space. More specifically, avoiding the technicality that each pixel element is bounded and discrete,

1. A grayscale image of resolution of $H \times W$ is a vector in $\mathbb{R}^H \otimes \mathbb{R}^W$.
2. An image with C channels of the same resolution is an element of $\mathbb{R}^C \otimes \mathbb{R}^H \otimes \mathbb{R}^W$.
3. A video with C channels and of the same resolution is an element of $\mathbb{R}^T \otimes \mathbb{R}^C \otimes \mathbb{R}^H \otimes \mathbb{R}^W$, where T is the time dimension which is usually represented in some discrete frames.

Since tensor product spaces are also vector space, there is nothing new we have to introduce. Everything we talked about so far applies to images when treated as vectors. In fact, so far we have been interpreting images of size (C, H, W) through the isomorphism

$$\phi : \mathbb{R}^C \otimes \mathbb{R}^H \otimes \mathbb{R}^W \to \mathbb{R}^{C \times H \times W} \tag{122}$$

that essentially "unravels" the image.

3.1 Convolutional and Pooling Layers

So far, we have seen the power of multilayer perceptrons and their predictive ability on moderately sized vectors. In fact, if we process the MNIST with a simple MLP of 2 layers and 512 nodes each, we can easily get 95% accuracy within 10 epochs. However, these MNIST pictures are extremely low resolution, at $1 \times 28 \times 28$, and for even moderately sized images we can that there is a huge blowup of parameters needed. [8] Clearly, this is not efficient, and so the only way to move on is to create a sparser representation of the network. This is where convolutional kernels come in (note that this is completely different than the kernels mentioned in supervised learning, with support vector machines and RKHS), introduced in [6].

Definition 3.2 (Convolutional Kernel)

A **convolution operator** on a vector space V representing an image space is simply a special type of linear map that is parameterized by a much smaller set of numbers, stored within a **kernel** or **filter**. In all honesty, it is much easier to go through examples to see how they work, so in this definition I will focus more on describing the hyperparameters. Given an image of shape (C, H, W), the convolution is essentially a sliding window that computes a dot product between the kernel and the window that the kernel covers over the image.

1. The sliding window size is (W_{ker}, H_{ker}), which is conventionally square but does not need to be.
2. This sliding window must compute over all channels, so in fact it is of shape $(C_{in}, W_{ker}, H_{ker})$. This would generate one output channel image.
3. Multiple kernels can be used concurrently to generate different channel images. Therefore, if we want to have a collection of C_{out} outputs that are extracted from each kernel, our total kernel would be a collection of C_{out} kernels of shape $(C_{in}, W_{ker}, H_{ker}, C_{out})$. Therefore, the total equation is

$$(X * K)_{f,i,j} \coloneqq \sum_c \sum_{p,q} X_{c,i+p,j+q} \cdot K_{c,p,q,f} + b_{f,i,j} \tag{123}$$

where c is the channel index, p, q are the location indices, f is the output channel index, and b is some bias term.

[8] For example, an RGB image that is $3 \times 1024 \times 1024$ would have 3m parameters, and then defining a dense linear map to even 1000 dimensions would take 3 billion parameters. Given that 32-bit floating point is 4 bytes, this already takes up 12GB of memory just to load the network.

4. The stride parameter s can also be set to determine the stride of the kernel $\mathcal{K}$.
5. Another thing to note is that the output image of a kernel would be slightly smaller than the input image, since the kernel cannot go over the edge. However, there are padding schemes to preserve the original dimensions.

From the equation above, we can see that a convolutional layer, assuming that it has full padding, is a linear map

$$\mathcal{K} : \mathbb{R}^{C_{in}} \otimes \mathbb{R}^H \otimes \mathbb{R}^W \to \mathbb{R}^{C_{out}} \otimes \mathbb{R}^H \otimes \mathbb{R}^W \tag{124}$$

The vector space of linear maps mapping between these two spaces has $C_{in}H^2W^2C_{out}$ dimensions, which is extremely large, but parameterizing $\mathcal{K}$ with this matrix reduces the set of relevant convolutional maps to a subspace that is $(1+C_{in}H_{ker}W_{ker})C_{out}$ dimensional (with the $+1$ due to a bias term, making this an affine map). This is essentially what a convolution is: sparse matrix multiplication, and there is nothing else that makes it different from a classical feedforward neural network. It's just computationally efficient matrix multiplication for high-dimensional vectors.

Just to explicitly see what is actually computed, let's do one computational example.

Example 3.1 (Tarokh, Duke ECE685)

Consider an RGB image $X = [X_0, X_1, X_2]$ with three channels, and given as follows

$$X_0 = \begin{bmatrix} 2 & 1 & 0 & 0 \\ 0 & 0 & 2 & 1 \\ 0 & 2 & 0 & 1 \\ 2 & 1 & 0 & 1 \end{bmatrix}, \quad X_1 = \begin{bmatrix} 2 & 2 & 0 & 0 \\ 0 & 0 & 2 & 1 \\ 0 & 0 & 2 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}, \quad X_2 = \begin{bmatrix} 2 & 1 & 0 & 0 \\ 0 & 0 & 2 & 1 \\ 2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} \tag{125}$$

The image is passed through the convolutional filter with the weights $W = [W_0, W_1, W_2] \in \mathbb{R}^{3\times3\times3}$ and step size 1, and given as follows

$$W_0 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -2 & 0 \\ 0 & 0 & -1 \end{bmatrix}, \quad W_1 = \begin{bmatrix} 1 & 2 & 0 \\ 2 & 0 & -1 \\ 0 & -1 & 1 \end{bmatrix}, \quad W_2 = \begin{bmatrix} 0 & 0 & -2 \\ 0 & 1 & 2 \\ -2 & 2 & 0 \end{bmatrix} \tag{126}$$

The output of the convolutional filter is given as

$$Y = \text{ReLU}\left(\sum_{i=0}^{2} (X_i' * W_i) + 2 \cdot 1_{4\times4} \right) \tag{127}$$

where Y is the output image, X' is the input image after applying 0 padding around the edges, and $*$ is the discrete convolution operator. Compute the output Y, and then apply max pooling on nonoverlapping 2×2 submatrices, and then apply average pooling on non-overlapping 2×2 submatrices.

Solution 3.1

We can compute

$$X_0 * W_0 = \begin{bmatrix} -4 & -4 & -1 & 0 \\ -2 & 2 & -4 & -2 \\ -1 & -4 & -1 & 0 \\ -4 & -2 & 2 & -2 \end{bmatrix}$$

$$X_1 * W_1 = \begin{bmatrix} -2 & 6 & 3 & -1 \\ 4 & 6 & -1 & 4 \\ 1 & -3 & 5 & 7 \\ -1 & 0 & 5 & 2 \end{bmatrix}$$

$$X_2 * W_2 = \begin{bmatrix} 4 & 1 & 4 & -2 \\ 2 & 0 & 4 & 11 \\ 2 & -2 & -4 & 2 \\ 2 & 1 & 2 & 1 \end{bmatrix}$$

and so we get

$$Y = \begin{bmatrix} 0 & 5 & 8 & 0 \\ 6 & 10 & 1 & 5 \\ 4 & 0 & 2 & 11 \\ 0 & 1 & 11 & 3 \end{bmatrix} \tag{128}$$

Maxpooling and average pooling gives us

$$\max(Y) = \begin{bmatrix} 10 & 8 \\ 4 & 11 \end{bmatrix} \text{ and } \operatorname{avg}(Y) = \begin{bmatrix} 21/4 & 7/2 \\ 5/4 & 27/4 \end{bmatrix} \tag{129}$$

In addition to computational efficiency and weight sharing, convolutional operators capitalize on the principle of **locality**, i.e. that pixels are directly related to adjacent pixels. For example, a pixel representing a portion of a dog's ear would not be related to the background, but the color and positioning should be related to the dog's face, which may be within a certain neighborhood around. This has been shown to be similar to the human visual system and is thus well motivated. Though this next topic has more to do with classical image processing than computer vision, there are a surprising number of features that these convolutional filters can extract from an image. By treating them as a discretized form of a partial derivative (as the vertical and horizontal edge detection) or as the Hessian operator (sharpening), we can extract many features from them.

Example 3.2 ()

Given the original image below, we show various convolutional filters applied on the image. Note that the kernel matrix may have the property that all of its entries sum to 1, meaning that on average, the expected value of the brightness of each pixel will be 0, and the values will be left unchanged on average. However, this is not a requirement.

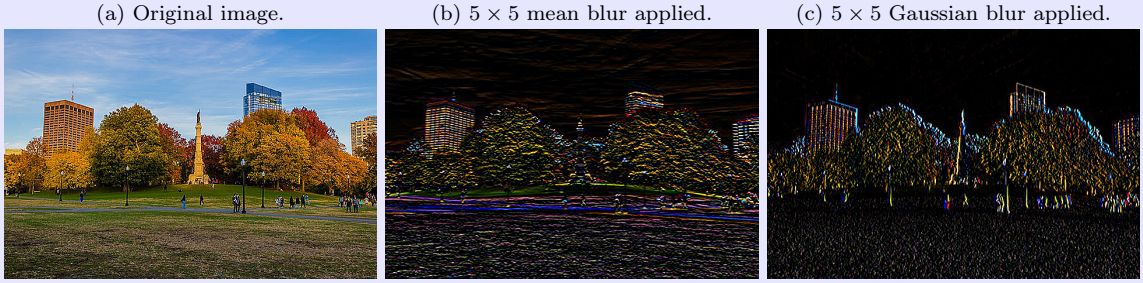
$$\text{Original} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{Mean} = \frac{1}{25}\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \quad \text{Gaussian} = \frac{1}{273}\begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}$$

$$\text{Sharpen} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} \quad \text{Horizontal} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad \text{Vertical} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

These filters visually output the following images. Note that these filters are each acting on the image

by acting individually on each channel and then combining the 3 outputs to create the new RGB image.

(a) Original image. (b) 5×5 mean blur applied. (c) 5×5 Gaussian blur applied.



(d) Sharpening kernel. (e) Horizontal edge detection. (f) Vertial edge detection.

Figure 7: Different convolutional kernels acting on the same image. Several useful features like edges can be detected with these simple linear maps.

We have seen in the example above that we can interpret each output channel of a convolution as a feature. That is, our original input image with $C_{in} = 3$ channels may go through a convolution that has $C_{out} = 10$ output channels, producing 10 grayscale images. Each of these images may represent a feature that is extracted from the image through a custom kernel. When we stack convolutional layers together (with nonlinearities in between, of course), we can produce more complicated transformations that extract more abstract features. For example, while the first layer or two may extract certain edges within a dog, perhaps the fourth or fifth convolutional layer will be able to detect the presence of ears. This is a hand-wavy example, but if you actually visualize the outputs of these layers during forward prop, it is possible to see this in action.

What we eventually hope for is that we can extract higher level features that can be encoded in moderate-dimensional vectors. Unfortunately, the rate at which regular convolutional filters (especially when there is padding and a stride of 1) does not shrink the resolution of the input images at a fast enough rate. For example, having a $3 \times 3 \times 3 \times 3$ kernel with no padding on a $(3, 100, 100)$ image will decrease the dimensions to only $(3, 98, 98)$ only. Therefore, we do some very simple operations to reduce the resolution faster.

Definition 3.3 (Pooling Layers)

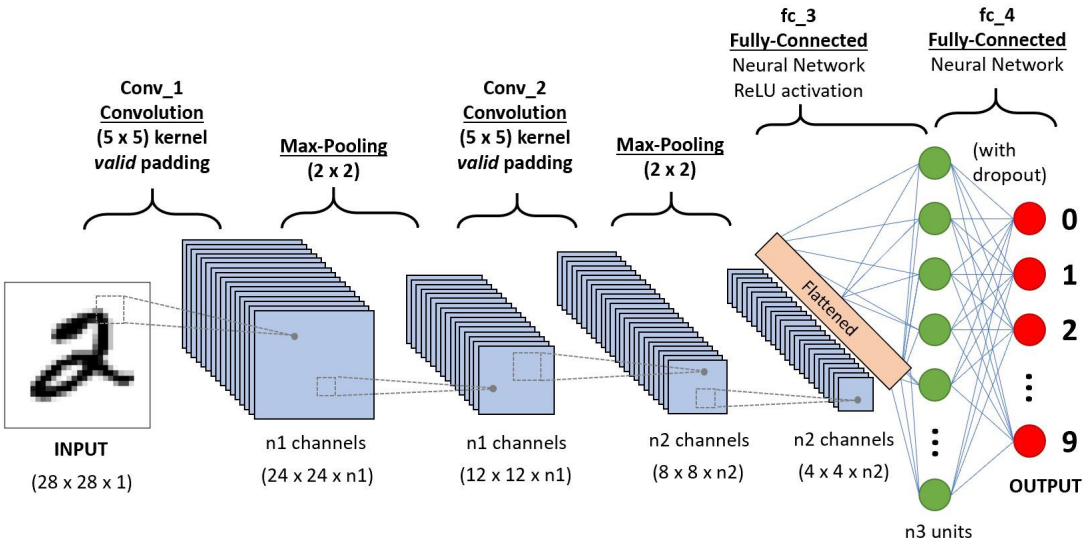
A pooling layer takes in an input image of dimension (C, H, W) and essentially does **downsampling** on it, involving some method of pooling local groups of pixels together into one value. There are several ways to do this:

1. **Max Pooling** refers to dividing each channel of the image into a "checkerboard" of $P \times P$ (where P is a hyperparameter and does not necessarily have to be a square) matrices and simply choosing the maximum pixel value from it.
2. **Average Pooling** is the same as max pooling but we just take the average.

Clearly, these are not expensive operations and are an effective way to downsample. Therefore, the same $(3, 100, 100)$ image, after one convolutional layer followed by a pooling layer, will result in a $(3, 49, 49)$ image.

Ultimately, after a series of convolutions and pooling, we would want to reduce this image to a form of (C, H, W), where both H and W are small and C is large. This is because for each value of C, say $C = 1$, the cross section $\{(1, W, H)\}$ would encode the value of the feature identified by C. In fact, it could be the fact that both H and W are 1, and $C = 10$. Then, we would essentially be looking at an array of 10 numbers, which could encode the presence of some abstract features. For example, the first value $C = 1$ would encode the presence of an eye, which in the end has a value of 0.9 (high probability), the second $C = 2$ could encode the presence of an ear, and so on.

Perhaps the sparsity of these maps may not allow the convolutional layers alone to extract all the features we need, so it is common to unwrap the features and then add a few fully connected layers at the end, which is much more computationally feasible now that the convolutions and pooling layers have reduced the dimensionality whilst extracting useful features with the concept of locality. This turns out to have comparable performance to regular MLPs with a fraction of the computational cost, and can easily reach 98% validation accuracy on the MNIST dataset.



3.2 Backpropagation

The fully connected layers are all taken care of, but now it's the convolutional layers and the pooling layers. The convolutional layers are also linear maps, so they can be treated the same way. However, the pooling layers may be nonlinear.

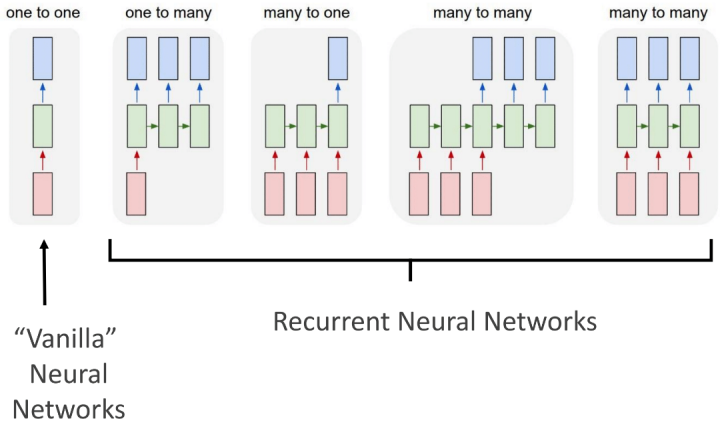
1. Average pooling is linear, so no worries here.
2. Max pooling is not linear, but it is the next best thing: piecewise linear.

3.3 Visualizing Activation Maps

3.4 CAM and Grad-CAM

4 Recurrent Neural Networks

Let's focus on what is lacking in the vanilla feedforward neural net architecture. In a vanilla feedforward neural net architecture, we had a one to one map, where we take an input of fixed size and we map it to an output of fixed size. Perhaps we would want a one-to-many model, which takes in an image for example and outputs a variable-length description of the image. Or a many-to-many (e.g. machine translation from a sequence of words to a sequence of words) or many-to-one. Just as a convolutional neural network is specialized for processing a grid a values such as an image, a recurrent neural network is specialized for processing a sequence of values (e.g. audio, video, text, speech, time series). It is not limited to a fixed size of inputs and outputs.



Now to build such a model where the input or output elements are unbounded, we must take advantage of weight sharing (as seen in the CNN architecture) to control the size of our neural net. Furthermore, the fact that we should take in a sequence of inputs means that we may want to introduce some recursive structure in our neural net. Consider the classical form of a dynamical system driven by an external signal $\mathbf{x}$ as

$$\mathbf{s}_t = f(\mathbf{s}_{t-1}, \mathbf{x}_t; \boldsymbol{\theta})$$

which defines a recurrent relationship. Similarly, we can write $\mathbf{h}$ to represent hidden neurons and write

$$\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{x}_t; \boldsymbol{\theta})$$

which indicates that the state of a hidden neuron is dependent on both the previous neuron and an input at time t. Through recursion, the hidden state $\mathbf{h}_t$ contains all information about the inputs $\mathbf{x}_1, \dots, \mathbf{x}_t$ in the form of a complex function $\mathbf{g}$.

$$\begin{aligned} \mathbf{h}_t &= \mathbf{g}_t\big(\mathbf{x}_t, \mathbf{x}_{t-1}, \dots, \mathbf{x}_1\big) \\ &= f(\mathbf{h}_{t-1}, \mathbf{x}_t; \boldsymbol{\theta}) \end{aligned}$$

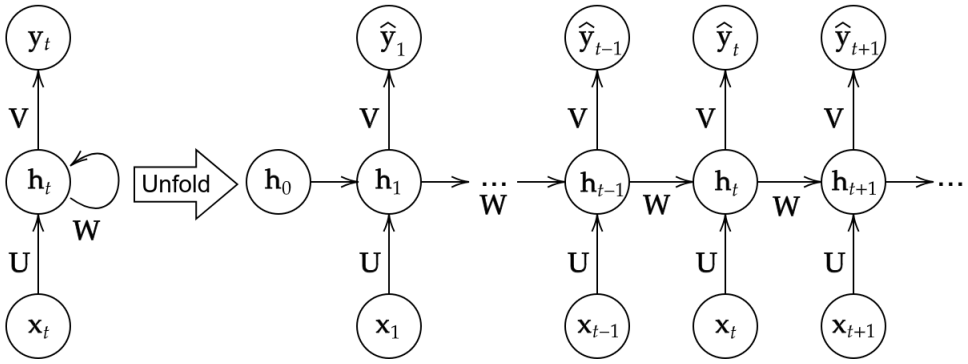
The fact that we can factorize $\mathbf{g}_t$ into a repeated application of function $\mathbf{f}$ gives us two advantages:

1. Regardless of the sequence length, the learned model always has the same input size because it is specified in terms of transition from one state to another state, rather than specified in terms of a variable-length history of states.
2. It is possible to use the same transition function f with the same parameters at every time step. Since we do not have a growing number of parameters to optimize as our sequential data grows, training an RNN is still computationally feasible.

These two factors make it possible to learn a single model f that operates on all time steps and all sequence lengths, rather than needing to learn a separate model $\mathbf{g}_t$ for all possible time steps.

4.1 Unidirectional RNNs

A single layer unidirectional RNN is a direct application of the idea mentioned in the previous section. We can first look at its computational graph



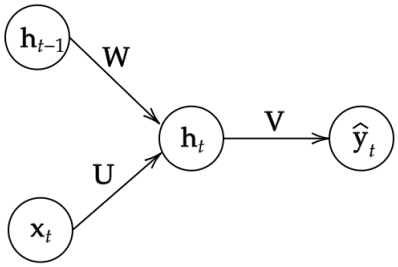
The activation functions that map to the hidden nodes and the outputs will be labeled $\boldsymbol{\sigma}_h$ and $\boldsymbol{\sigma}_y$, respectively. In general the W will represent the left and right mappings between hidden nodes, the U will represent the map going up from the input or hidden node to a hidden node, and V is the final mapping from a hidden node to an output. We only label the arrows with the matrices, though a bias term and the nonlinear activation function are still there. That is, we can summarize our network as

$$\begin{aligned}\mathbf{h}_t &= \mathbf{f}(\mathbf{h}_{t-1}, \mathbf{x}_t; \boldsymbol{\theta}) = \boldsymbol{\sigma}_h\big(\mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t + \mathbf{b}_h\big) \\ \mathbf{y}_t &= \boldsymbol{\sigma}_y\big(\mathbf{V}\mathbf{h}_t + \mathbf{b}_y\big)\end{aligned}$$

for $t = 1, \dots, \tau$, where $\mathbf{h}_0$ is initialized to be zeroes or some small vector. The dimensions of the maps and the variables are listed for clarification:

1. $\mathbf{x}_t \in \mathbb{R}^d$ for all t
2. $\mathbf{h}_t \in \mathbb{R}^h$ for all t
3. $\mathbf{b}_h \in \mathbb{R}^h$
4. $\mathbf{U} \in \mathbb{R}^{h \times d}$
5. $\mathbf{W} \in \mathbb{R}^{h \times h}$

As we can see, the hidden node from the previous time step provides a form of memory, or context, that encodes earlier processing and informs the decisions to be made at later points in time. Adding this temporal dimension makes RNNs appear to be more complex than non-recurrent architectures, but in reality, they're not all that different. Consider the rearranged architecture of an RNN below.



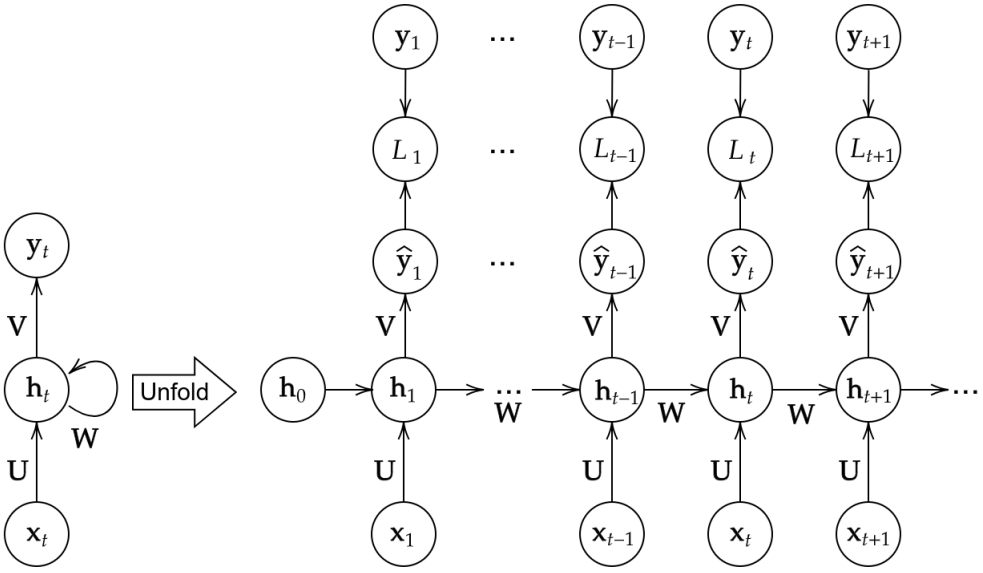
4.1.1 Loss Functions

The form of the loss for a RNN will have to be slightly modified, since we can have multiple outputs. If we have a given input-output pair $\mathbf{x}^{(n)}, \mathbf{y}^{(n)}$, and we are interested producing a single output, then this is similar to what we already do with regular NNs. If we are interested in producing a sequence of outputs,

then we can average the loss functions individually so that equal weight is placed on the prediction at each relevant timestep. This is called

$$L = \frac{1}{|T|} \sum_{t \in T} L_t$$

Sometimes, even with single inputs it may be good to include other intermediate terms in the loss so that we can direct the neural net to converge faster to what the correct answer should be.



Note that one problem is that the errors can build up as the RNN predicts outcomes. For example, if we predicted $\mathbf{x}_1 \mapsto \hat{\mathbf{y}}_1$ we can compute the loss as $L_1(\mathbf{y}_1, \hat{\mathbf{y}}_1)$. However, there are two ways to compute the second loss: with inputs $L_2(\mathbf{x}_1, \mathbf{x}_2)$ or with $L_2(\mathbf{x}_1, \hat{\mathbf{y}}_1)$. One just uses the ground truth while the other uses the previous prediction for the next prediction, which can accumulate error. Both ways are feasible for loss computation, but it is generally done in the former way, called **teacher forcing**. This is analogous to a human student taking a multi-part exam where the answer to each part depends on the answer to the preceding part. Rather than grading every answer in the end, with the risk that the student fails every single part even though they only made a mistake in the first one, a teacher records the score for each individual part and then tells the student the correct answer, to be used in the next part.

4.1.2 Backpropagation Through Time

Now if we wanted to backpropagate through this RNN, we can compute

$$\frac{\partial L_t}{\partial \mathbf{W}} = \frac{\partial L_t}{\partial \hat{\mathbf{y}}_t} \frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}}$$

where the first term depends on the specific form of the loss and the second is simply the matrix $\mathbf{V}$. This all looks the same as backpropagation for a MLP, but since $\mathbf{W}_{hh}$ is used at multiple layers, we can reduce the third term in the equation to

$$\frac{\partial L_t}{\partial \mathbf{W}} = \frac{\partial L_t}{\partial \hat{\mathbf{y}}_t} \frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{h}_t} \left(\sum_{k=1}^{t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} \frac{\partial \mathbf{h}_k}{\partial \mathbf{W}} \right)$$

where

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} = \prod_{i=k+1}^{t} \frac{\partial \mathbf{h}_i}{\partial \mathbf{h}_{i-1}}$$

is computed as a multiplication of adjacent time steps. Now this can be very problematic, since if we have a lot of multiplications, then depending on the randomness of these matrices the gradient may be highly

unstable, causing the vanishing or exploding gradient problem. We can elaborate on this a little further. Note that the hidden linear maps are known to be square matrices. We can expand out the derivative without the constant terms on the left as such:

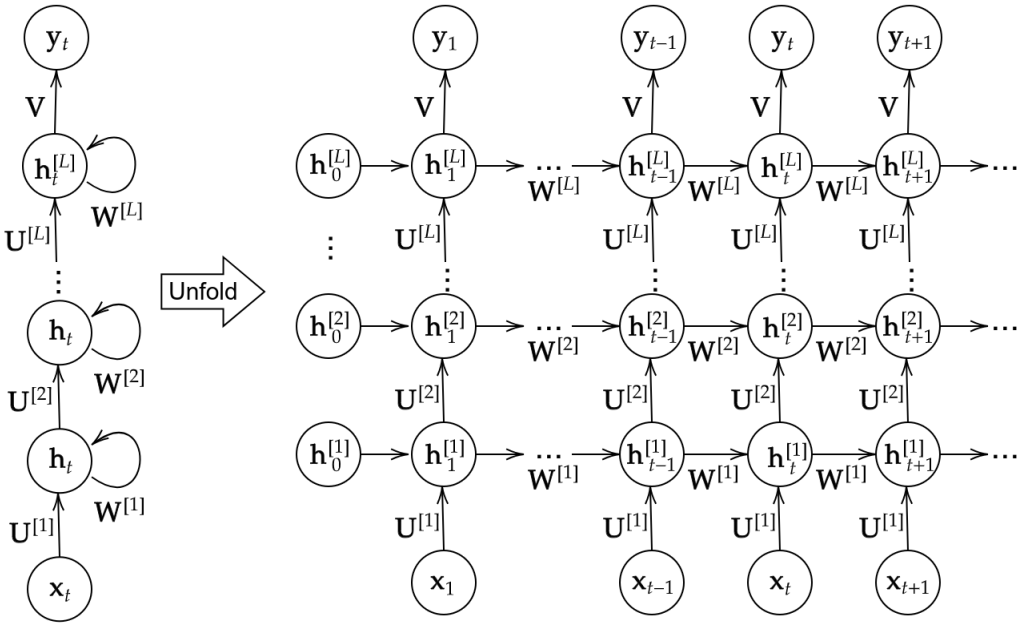
$$\sum_{k=1}^{t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} \frac{\partial \mathbf{h}_k}{\partial \mathbf{W}} = \sum_{k=1}^{t} \prod_{k<i\leq t} \frac{\partial \mathbf{h}_i}{\partial \mathbf{h}_{i-1}} \frac{\partial \mathbf{h}_k}{\partial \mathbf{W}}$$

and we can see if at some point one of the $\frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}}$ tend to be small just from randomness, then their product for all coefficients where $k \leq j$ will be small too. This means that all the information, or memory, from the jth hidden state and before will vanish. In fact, if the spectrum (the set of eigenvalues and eigenvectors) is less than 1, then the multiplication of these derivatives will converge to a 0 matrix, and so we have an exponential memory loss throughout the network.

Furthermore, we can compute these gradients in batches by splitting up the corpus into several sentences, and sampling the sentences for gradient computation. Therefore, a forward or backward pass has a runtime complexity of $O(\tau)$ and cannot be reduced by parallelization because the forward propagation graph is inherently sequential. Each time step may only be computed after the previous one. States computed in the forward pass must be stored until they are reused during the backward pass, so the memory cost is also $O(\tau)$.

4.1.3 Stacked Unidirectional RNNs

Note that since we really have three matrices to optimize in the regular RNN, this may not be so robust. Therefore, we would like more hidden layers to capture further nonlinearities in an RNN, which is why we introduce a **stacked RNN** as shown below:



Now in this case, there are more layers of hidden nodes that an input must go through before it reaches the output node. We can expand out the computations as such, for $t = 1, \dots, \tau$, $l = 2, \dots L$:

$$\begin{aligned}
\mathbf{h}_t^{[1]} &= \boldsymbol{\sigma}_h\big(\mathbf{W}^{[1]}\mathbf{h}_{t-1}^{[1]} + \mathbf{U}^{[1]}\mathbf{x}_t + \mathbf{b}_h^{[1]}\big) \\
\mathbf{h}_t^{[l]} &= \boldsymbol{\sigma}_h\big(\mathbf{W}^{[l]}\mathbf{h}_{t-1}^{[l]} + \mathbf{U}^{[l]}\mathbf{x}_t + \mathbf{b}_h^{[l]}\big) \\
\mathbf{y}_t &= \boldsymbol{\sigma}_y\big(\mathbf{V}\mathbf{h}_t^{[L]} + \mathbf{b}_y^{[L]}\big)
\end{aligned}$$

or we could get rid of the first equation all together if we set $\mathbf{x}_t = \mathbf{h}_t^{[0]}$. Note that the hidden nodes $\mathbf{h}_t^{[l]}$ for all t and all $l \neq 0$ are all in $\mathbb{R}^h$, i.e. all hidden nodes will be h-dimensional. Therefore, most of the parameter matrices that we work with are square: $\mathbf{W}^{[l]} \in \mathbb{R}^{h \times h}$ and $\mathbf{U}^{[l]} \in \mathbb{R}^{h \times h}$ except for $\mathbf{U}^{[1]} \in \mathbb{R}^{h \times d}$.

4.2 Bidirectional RNNs

4.2.1 PyTorch Implementation

The implementation in PyTorch actually uses *two* bias terms $\mathbf{b}_{hW}^{[l]}$ and $\mathbf{b}_{hU}^{[l]}$ rather than just $\mathbf{b}_h^{[l]}$. This is technically not needed since the bias terms will just cancel out, but this is just how cuDNN (Cuda Deep Neural Network) is implemented.

$$\mathbf{h}_t^{[l]} = \boldsymbol{\sigma}_h\big(\mathbf{W}^{[l]}\mathbf{h}_{t-1}^{[l]} + \mathbf{b}_{hW}^{[l]} + \mathbf{U}^{[l]}\mathbf{x}_t + \mathbf{b}_{hU}^{[l]}\big)$$

Let us look at a 2 layer RNN of sequence length 5. The input features will be set to 10, meaning that each $\mathbf{x} \in \mathbb{R}^{10}$. The hidden nodes will all be in $\mathbb{R}^{20}$.

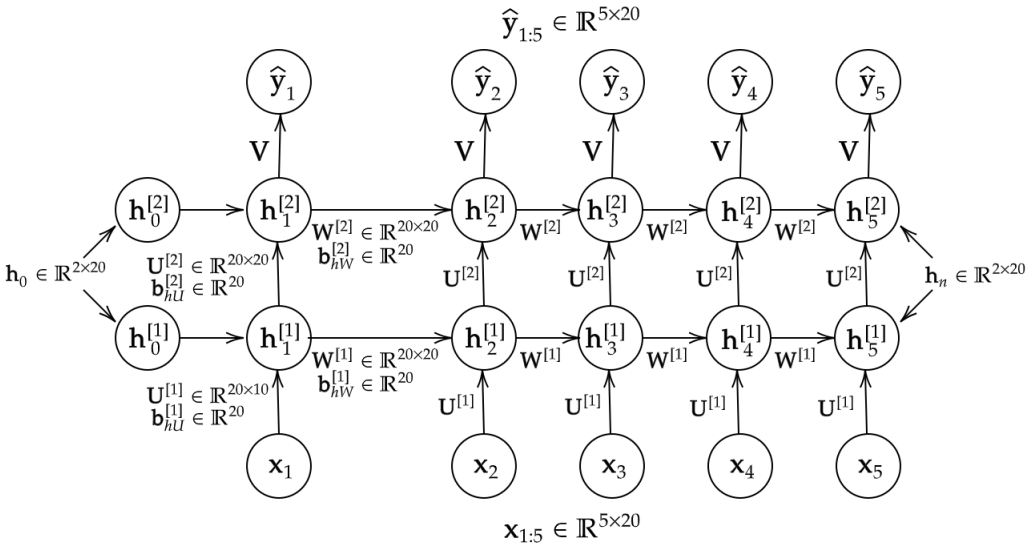
```
input_features = 10
hidden_features = 20
num_layers = 2
sequence_length = 5

rnn = nn.RNN(input_features, hidden_features, num_layers)
input = torch.randn(sequence_length, input_features)
h0 = torch.randn(num_layers, hidden_features)
print(input.size(), h0.size())
# torch.Size([5, 10]) torch.Size([2, 20])

print([weight.data.size() for weights in rnn.all_weights for weight in weights])
# [torch.Size([20, 10]), torch.Size([20, 20]), torch.Size([20]), torch.Size([20]),
torch.Size([20, 20]), torch.Size([20, 20]), torch.Size([20]), torch.Size([20])]

output, hn = rnn(input, h0)
print(output.size(), hn.size())
# torch.Size([5, 20]) torch.Size([2, 20])
```

The corresponding diagram is shown below.



As we expect, there are 8 vectors/matrices we must optimize: $\mathbf{W}^{[1]}, \mathbf{W}^{[2]}, \mathbf{U}^{[1]}, \mathbf{U}^{[2]}, \mathbf{b}_{hU}^{[1]}, \mathbf{b}_{hW}^{[1]}, \mathbf{b}_{hW}^{[2]}, \mathbf{b}_{hU}^{[2]}$.

4.3 Long Short Term Memory (LSTMs)

In theory, RNNs are very beautiful and can be applied in all cases, but in practice they do not perform very well, mainly due to the vanishing/exploding gradient problem.

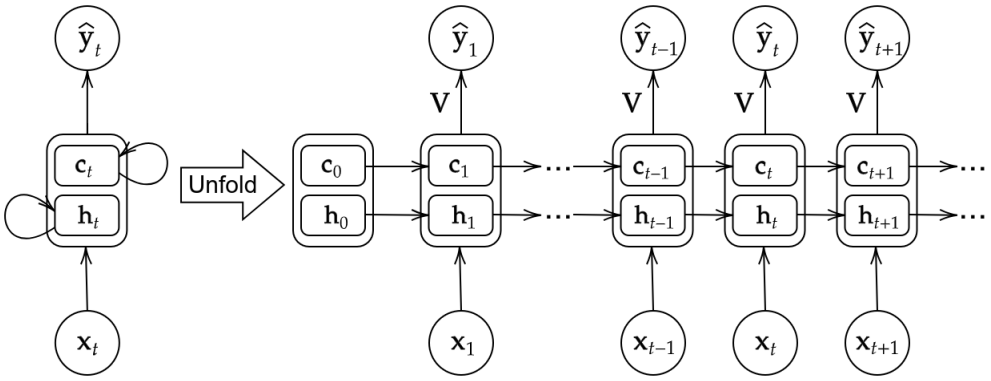
1. An exploding gradient is easy to fix, since we can just use the max-norm regularization, i.e. **gradient clipping**, to just set a max vamlue for the gradients if they grow too large.
2. The **truncated backpropagation through time** (TBPTT) simply limits the number of times steps the signal can backpropagate after each forward pass, e.g. even if the sequence has 100 time steps, we may only backpropagate through 20 or so.
3. The **LSTM** model uses a memory cell for modeling long-range dependencies and avoids the vanishing gradient problems.

Historically LSTMs were used in achieving state-of-the-art results in 2013 through 2015, in taks susch as handwriting recognition, speech recognition, machine translation, parsing, and image captioning, as well as language models. They became to dominant approach for most NLP tasks, but in 2021, they have been overshadowed by transformer models, which we will talk about next.

LSTMs have a much more complicated unit to work with, so let's go through it slowly. Note that so far, a one-layer RNN consisted of recursive mappings of the form

$$(\mathbf{x}_t, \mathbf{h}_{t-1}) \mapsto (\mathbf{h}_t, \hat{\mathbf{y}}_t)$$

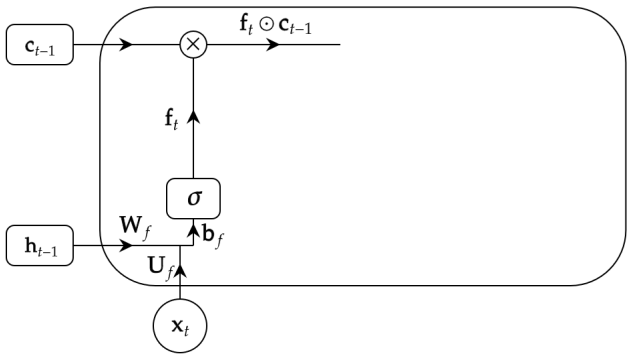
We can interpret the vector $\mathbf{h}_{t-1}$ as the **short term memory**, or **hidden state**, that contains information used to predict the next output value. However, this can be corrupted (e.g. forgetting information from many steps ago), so we add an additional **long term memory**, or **cell state**, vector $\mathbf{c}_t$ that should be preserved. Therefore, we have two arrows coming out of each hidden layer, as shown below in the one-layer LSTM.



The mechanisms of the cell is quite complex, but the three basic steps are: (1) we forget a portion of the long term memory, (2) we add new long term memory, (3) we add new short term memory. Let us demonstrate this step by step. We are given three inputs: the previous long-term memory $\mathbf{c}_{t-1}$, the previous short-term memory $\mathbf{h}_{t-1}$, and the input at current time $\mathbf{x}_t$. In LSTMs, we only use the sigmoid and tanh activation functions, so we will denote them explicitly as $\boldsymbol{\sigma}$ and tanh. For clarity, we will not write the matrix operations in the diagram anymore.

1. The **forget gate** (denoted by $\mathbf{f}$) takes an affine combination of $\mathbf{h}_{t-1}$ and $\mathbf{x}_t$ and puts it through the sigmoid activation function to generate a vector $\mathbf{f}_t$ that has every element in $(0, 1)$. Then it element-wise multiplies it with $\mathbf{c}_{t-1}$, which essentially "forgets" a portion of the long-term memory.

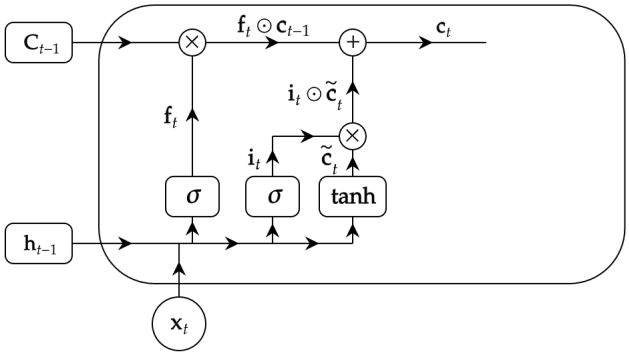
$$\mathbf{f}_t = \boldsymbol{\sigma}(\mathbf{W}_f \mathbf{h}_{t-1} + \mathbf{U}_f \mathbf{x}_t + \mathbf{b}_f)$$



2. The **input gate** (denoted by $\mathbf{i}$) consists of two activations with the following operations.

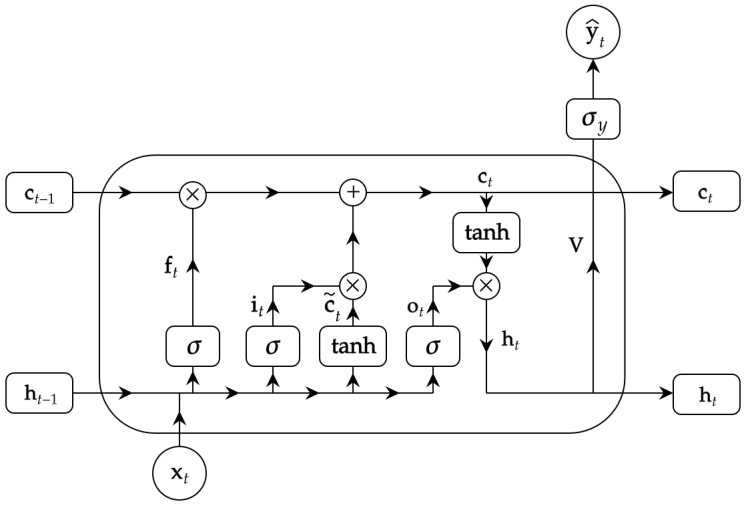
$$\begin{aligned}\mathbf{i}_t &= \boldsymbol{\sigma}(\mathbf{W}_i\mathbf{h}_{t-1} + \mathbf{U}_i\mathbf{x}_t + \mathbf{b}_i)\\ \tilde{\mathbf{c}}_t &= \mathbf{tanh}(\mathbf{W}_c\mathbf{h}_{t-1} + \mathbf{U}_c\mathbf{x}_t + \mathbf{b}_c)\\ \mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t\end{aligned}$$

The layer $\mathbf{i}$ can be seen as the filter that selects which information can pass through it and what information to be discarded. To create this layer, we pass the short-term memory and current input into a sigmoid function, which will transform the values to be between 0 and 1, indicating which information is unimportant. The second layer $\tilde{\mathbf{c}}$ takes the short term memory and current input and uses the tanh to transform the elements to be in $(-1, 1)$, which allows us to add or subtract the necessary information from the long term memory.



3. The **output gate** (denoted by $\mathbf{o}$) consists of two activations with the following operations. This again creates a separate filter that selects the relevant information needed for the short term memory.

$$\begin{aligned}\mathbf{o}_t &= \boldsymbol{\sigma}(\mathbf{W}_o\mathbf{h}_{t-1} + \mathbf{U}_o\mathbf{x}_t + \mathbf{b}_o)\\ \mathbf{h}_t &= \mathbf{o}_t \odot \mathbf{tanh}(\mathbf{c}_t)\\ \hat{\mathbf{y}}_t &= \boldsymbol{\sigma}_y(\mathbf{V}\mathbf{h}_t + \mathbf{b}_y)\end{aligned}$$



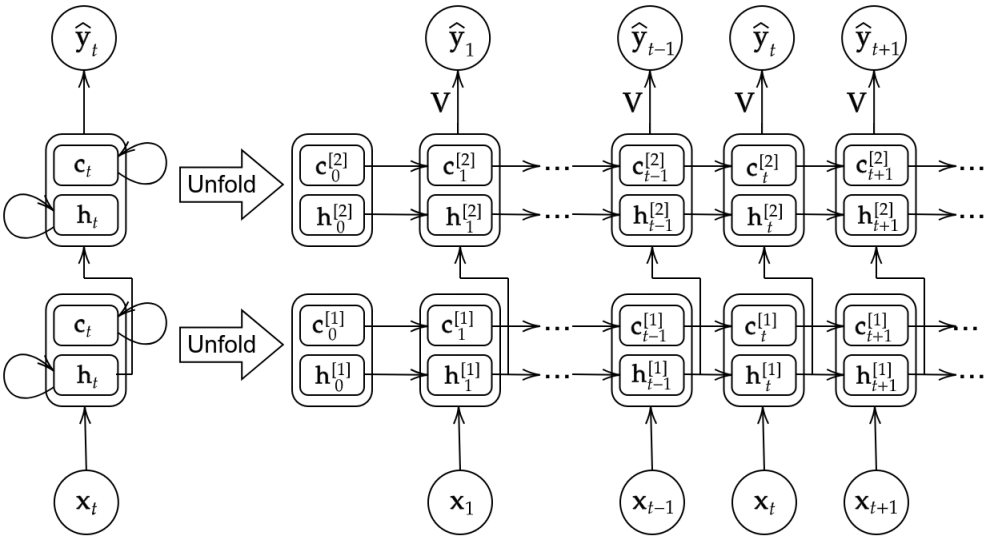
That is it! Now focusing on the cell state in the diagram above. Note that in order to go from cell state $\mathbf{c}_{t-1}$ to $\mathbf{c}_t$, there was not a whole lot done to it. We really just multiply it once, which potentially deletes some content, and add it once, which adds new content, and we are done. The magic is this addition, since unlike multiplication, which can result in an exponential decay of knowledge, you are just constantly adding new numbers to update the storage, allowing the cell state to behave much more like RAM of a computer.

The LSTM architecture also makes it easier for the RNN to preserve information over many timesteps. For example, if the forget gate $\mathbf{f}_t$ is set to $\mathbf{1}$ and the input gate set to $\mathbf{0}$, then the information of that cell is preserved indefinitely. In contrast, it's harder for a vanilla RNN to learn a recurrent weight matrix $\mathbf{W}$ that preserves information in the hidden state. In practice, a vanilla RNN would preserve memory up to maybe 7 timesteps (and increasing this is extremely difficult) while a LSTM would get about 100 timesteps, so in practice you should almost always just use a LSTM.

Unfortunately, LSTM doesn't *guarantee* that there is no vanishing or exploding gradients, but it does provide an easier way for the model to learn long-distance dependencies. Note that the gradient problem is not just a problem for RNNs; any neural architecture (including a feed-forward or convolutional) with very deep layers with multiple compositions of functions may suffer. Due to the chain rule and choice of nonlinearity function, these gradients can become vanishingly small and lower layers are learned very slowly. However, we can still implement residual connections to allow for more gradient flow such as ResNet, DenseNet, and HighwayNet.

4.3.1 Multilayer LSTMs

We can extend this architecture in the exactly same way for multilayer LSTMs. Note that we should be careful of the transformations each arrow represents. For the arrows going from $\mathbf{h}_t^{[l]} \mapsto \mathbf{h}_t^{[l+1]}$, there is no further transformation since we are just pushing this vector as an input to the next LSTM node. However, the arrow pushing from $\mathbf{c}_t^{[L]} \mapsto \hat{\mathbf{y}}_t$ does have an extra affine transformation with $\mathbf{V}$ and $\mathbf{b}_y$, followed by some link function $\boldsymbol{\sigma}_y$ before we have the true prediction.



This follows the recursive equations, with $\mathbf{x}_t = \mathbf{h}_t^{[0]}$.

$$\text{Forget Gate } \begin{cases} \mathbf{f}_t^{[l]} = \boldsymbol{\sigma}(\mathbf{W}_f^{[l]}\mathbf{h}_{t-1}^{[l]} + \mathbf{U}_f^{[l]}\mathbf{h}_t^{[l-1]} + \mathbf{b}_f^{[l]}) \end{cases}$$

$$\text{Input Gate } \begin{cases} \mathbf{i}_t^{[l]} = \boldsymbol{\sigma}(\mathbf{W}_i^{[l]}\mathbf{h}_{t-1}^{[l]} + \mathbf{U}_i^{[l]}\mathbf{h}_t^{[l-1]} + \mathbf{b}_i^{[l]}) \\ \tilde{\mathbf{c}}_t^{[l]} = \mathbf{tanh}(\mathbf{W}_c^{[l]}\mathbf{h}_{t-1}^{[l]} + \mathbf{U}_c^{[l]}\mathbf{h}_t^{[l-1]} + \mathbf{b}_c^{[l]}) \\ \mathbf{c}_t^{[l]} = \mathbf{f}_t^{[l]} \odot \mathbf{c}_{t-1}^{[l]} + \mathbf{i}_t^{[l]} \odot \tilde{\mathbf{c}}_t^{[l]} \end{cases}$$

$$\text{Output Gate } \begin{cases} \mathbf{o}_t^{[l]} = \boldsymbol{\sigma}(\mathbf{W}_o^{[l]}\mathbf{h}_{t-1}^{[l]} + \mathbf{U}_o^{[l]}\mathbf{h}_t^{[l-1]} + \mathbf{b}_o^{[l]}) \\ \mathbf{h}_t^{[l]} = \mathbf{o}_t^{[l]} \odot \mathbf{tanh}(\mathbf{c}_t^{[l]}) \end{cases}$$

$$\text{Output } \begin{cases} \hat{\mathbf{y}}_t = \boldsymbol{\sigma}_y(\mathbf{V}\mathbf{h}_t^{[L]} + \mathbf{b}_y) \end{cases}$$

where

1. $\mathbf{x}_t \in \mathbb{R}^d$ for all t
2. $\mathbf{f}_t^{[l]}, \mathbf{i}_t^{[l]}, \mathbf{o}_t^{[l]} \in (0,1)^h$
3. $\mathbf{h}_t^{[l]}, \tilde{\mathbf{c}}_t^{[l]} \in (-1,1)^h$
4. $\mathbf{c}_t^{[l]} \in \mathbb{R}^h$

and we must optimize the parameters

$$(\mathbf{W}_f^{[l]}, \mathbf{U}_f^{[l]}, \mathbf{b}_f^{[l]}),\ (\mathbf{W}_i^{[l]}, \mathbf{U}_i^{[l]}, \mathbf{b}_i^{[l]}),\ (\mathbf{W}_c^{[l]}, \mathbf{U}_c^{[l]}, \mathbf{b}_c^{[l]}),\ (\mathbf{W}_o^{[l]}, \mathbf{U}_o^{[l]}, \mathbf{b}_o^{[l]})$$

for $l = 1, \dots, L$. The fact that a LSTM uses the long term memory, in addition to the short term memory and the input, allows each cell to regulate the information to be kept or discarded at each time step before passing on the long-term and short-term information to the next cell. They can be trained to selectively remove any irrelevant information.

4.4 Gated Recurrent Units

5 Encoder-Decoder Models

Encoder decoder models refer to a model consisting of two neural nets: the encoder that takes in the input and maps it to some lower-dimensional vector. Then, the decoder takes in this encoded vector and attempts to use it to decode what we're trying to get. The type of neural network can be any: MLP, CNN, or RNN, depending on what problem you're trying to achieve.

Now, why would we want to do something like encode the input into some lower dimensional setting, and then have the decoder neural net extract what we want? It seems like we're making the problem harder. There are two reasons:

1. The input vector may not be in the correct form that we want. This is the motivation for the *seq2seq* model, where we are working with sequences of vectors that suffer from the problem of *locality* in RNNs. Therefore, it is necessary to encode this entire sequence into one vector, at the loss of dimension.
2. The input vector may be noisy or too high-dimensional itself. In CNNs, we saw that convolutional layers or pooling layers allow us to reduce the dimension to extract meaningful features from it. Likewise, we can train the encoder to extract useful features into a lower dimensional space, and then the decoder can efficiently work with this representation. This motivates the use of **autoencoders**, which can be done with MLPs, CNNs, or even RNNs.

Note that while these two algorithms fall in the paradigm of encoder-decoder networks, the seq2seq model is supervised while the autoencoder is unsupervised. In the seq2seq model, which deals with things like machine translation, we have a labeled dataset of sentences in language A corresponding with sentences in language B. However, in autoencoders, what we do is take a sample x from our dataset and use it both as the input and output to train our network. Since there is no additional labeling required, this is an unsupervised learning technique.

5.1 Autoencoders

Autoencoders are a type of unsupervised learning. We only use the inputs $\mathbf{x}_t$ for learning. We want to automatically extract meaningful features for the data and leverage the availability of unlabeled data. It can be used for visualization and compression. We can also build generative models with autoencoders.

Definition 5.1 (Autoencoder)

An autoencoder is a feed-forward neural net whose job is to take an input $\mathbf{x}$ and output $\hat{\mathbf{x}}$. It consists of an encoder $E_\phi : \mathcal{X} \to \mathcal{Z}$ and decoder $D_\theta : \mathcal{Z} \to \mathcal{X}$, where $\mathcal{X}$ is the input/output space and $\mathcal{Z}$ is the latent feature space.

1. The encoder model transforms $\mathbf{x}$ to a latent feature representation $\mathbf{z}$. It is a feed-foward, buttom-up neural net.
2. The decoder model maps $\mathbf{z}$ to a reconstruction $\hat{\mathbf{x}}$. It is generative, top-down.

I want to train the whole neural network such that the error between $\mathbf{x}$ and $\hat{\mathbf{x}}$ is minimized. We can consider a squared-error, for example.

$$\mathcal{L}(\mathbf{x}, \hat{\mathbf{x}}) = \frac{1}{2}||\mathbf{x} - \hat{\mathbf{x}}||_2^2$$

In the totally linear case, we have PCA. Some input $\mathbf{x} \in \mathcal{X} = \mathbb{R}^d$ is mapped to a smaller-dimensional $\mathcal{Z} = \mathbb{R}^k$.

$$\mathbf{x} \xrightarrow{V} \mathbf{z} \xrightarrow{U} \hat{\mathbf{x}}$$

and so the "network" essentially computes $\hat{\mathbf{x}} = UV\mathbf{x}$. Obviously the fact that $k < d$ is essential, since if $k \geq d$ then we can choose U and V such that $UV = I$, which is trivial.

This can be used for the following problem: Given m points $\mathbf{x}_1, \dots, \mathbf{x}_m \in \mathbb{R}^d$ and target dimension $k < d$, find the best k-dimensional subspade approximating the data. Formally, we want to find the matrices $U \in \mathbb{R}^{d \times k}$

and $V \in \mathbb{R}^{k \times d}$ that minimizes

$$f(U, V) = \sum_{i=1}^{m} ||\mathbf{x}_i - UV\mathbf{x}_i||_2^2$$

where V is the *compressor* and U is the *decompressor*. Now unfortunately, this loss f is not convex, though $f(U, \cdot)$ and $f(\cdot, V)$ are both convex.

Theorem 5.1 ()

We claim that the optimal solution is achieved when $U = V^T$ and $U^T U = I$.

Proof.

For any U, V, the linear map $\mathbf{x} \mapsto UV\mathbf{x}$ has a range R that forms a subspace of dimension k. Let $w_1, \dots, w_k$ be an orthonormal basis for R, which we arrange into columns of W. Hence, for each x_i there is $z_i \in \mathbb{R}^k$ such that $UVx_i = Wz_i$. Note that by construction,$W^T W = I$. Now we want to find out which z minimizes $f(x_i, z) = ||x_i - Wz||_2^2$. We know that for all $x \in \mathbb{R}^d, z \in \mathbb{R}^k$,

$$f(x, z) = ||x||_2^2 + z^T W^T W z - 2z^T W^T x = ||x||_2^2 + ||z||_2^2 - 2z^T W^T x$$

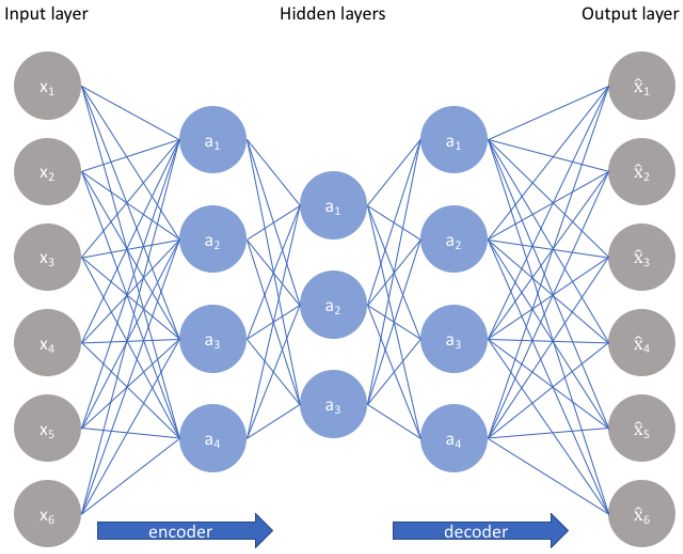
We want to minimize w.r.t. to z, so by taking the derivative and setting to 0, we get $z = W^T x$. This means that

$$\sum_{i=1}^{m} ||x_i - UVx_i||^2 \geq \sum_{i=1}^{m} ||x_i - UVx_i||^2 \geq \sum_{i=1}^{m} ||x_i - WW^T x_i||^2$$

and since U, V are optimal, equality is achieved and so instead of U, V, we can take W, W^T, with $WW^T x$ being the orthogonal projection of x onto R.

One application of PCA is eigenfaces, which assumes that the set of all faces (projected onto an image) approximately lies in a hyperplane.

Now let's go back to autoencoders, the nonlinear generalization of PCA. We can have several architectures, with none, one, or both the encoder/decoder having nonlinear activitation functions. Here is one architecture.



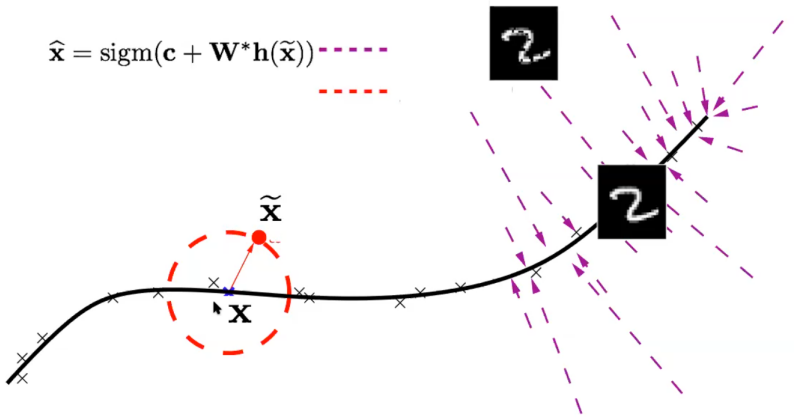
where we have

$$\text{Encoder} : \mathbf{h}(\mathbf{x}) = g(\mathbf{a}(x)) = \sigma(\mathbf{b} + \mathbf{W}\mathbf{x})$$
$$\text{Decoder} : \hat{\mathbf{a}}(\mathbf{x}) = \sigma(\mathbf{c} + \mathbf{W}^* \mathbf{h}(\mathbf{x}))$$

The parameter gradients are obtained by backpropagating the gradient $\nabla_\theta \mathcal{L}$ like a regular network, but if we force tied weights (i.e. $W^* = W^T$), then $\nabla_{\mathbf{W}} \mathcal{L}$ is the sum of two gradients. This is because $\mathbf{W}$ is present both in the encoder and decoder.

There are three things we can do to extract meaningful hidden features:

1. **Undercomplete Representation**: Make the latent dimension small. It compresses the input, but it may only be good for the training distribution and may not be robust to other types of input. If it is overcomplete, there is no guarantee that we will extract meaningful features.
2. **Denoising Autoencoder**: Injecting noise to the input. The idea is that the representation should be robust to the introduction of noise. We take the original input $\mathbf{x}$ and we randomly assign a subset of the inputs to 0, with probability ν, similar to dropout, to get our noisy input $\tilde{\mathbf{x}}$. Then we train the autoencoder with the loss comparing the output $\hat{\mathbf{x}}$ to the original, un-noisy input $\mathbf{x}$. We can do this for Gaussian additive noise too. As the visual below suggests, we are essentially "pushing" out inputs away from the manifold and training the autoencoder to denoise it, pulling it back.



3. **Contractive Autoencoder**: If we have the latent dimension greater than the input, then we can just add an explict term in the loss that penalizes that solution (e.g. promoting sparsity). For example, we can have the loss be

$$\mathcal{L}(f(\mathbf{x}^{(t)}) + \lambda ||\nabla_{\mathbf{x}^{(t)}} \mathbf{h}(\mathbf{x}^{(t)})||_F^2$$

where

$$||\nabla_{\mathbf{x}^{(t)}} \mathbf{h}(\mathbf{x}^{(t)})||_F^2 = \sum_{j,k} \left(\frac{\partial h(\mathbf{x}^{(t)})_j}{\partial x_k^{(t)}} \right)^2$$

which forces the encoder to throw away information. If one of the elements are 0, then we know that the kth element of the input has no effect on the jth element of the encoded output. Therefore, it tries to throw away as many elements of $\mathbf{x}$ as possible since the identity matrix will have a large Frobenius norm, essentially contracting the input representation.

We can also promote sparsity by adding a L1 penalty, forcing the feature space to be sparse.

The **predictive sparse decomposition** shows that the loss should be

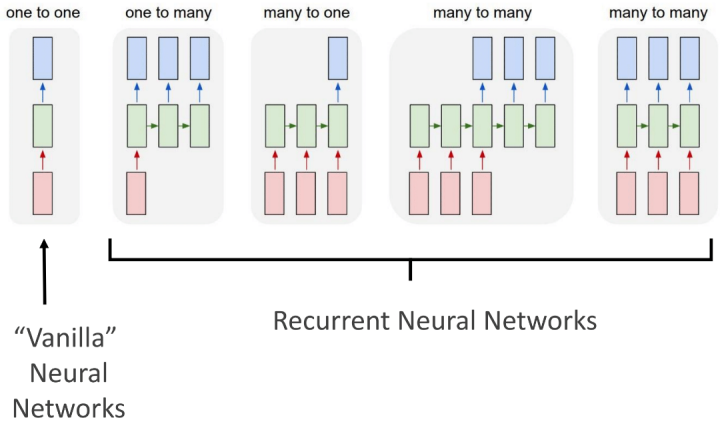
$$\min_{W, W^*, \mathbf{z}} ||W^*\mathbf{z} - \mathbf{x}||_2^2 + \lambda |\mathbf{z}|_1 + ||\sigma(W\mathbf{x}) - \mathbf{z}||_2^2$$

where the first term tells the decoder to reconstruct the original input well, the second tells the latent vector to be sparse, and the third tells us that we shouldn't lose too much information when we encode.

We could also have **stacked autoencoders**, with each layer of latent features having some desired sparsity.

5.2 Sequence to Sequence

We have mentioned that RNNs and LSTMs have the advantage of mapping from variable length inputs to variable length outputs. This can be done for any length input and any length output.



However, the RNN has the problem of *locality*, that the words next to the current word have a greater effect, and we are trying to generate sequences on the fly by reading in each word. Even for bidirectional RNNs, where we go through the whole sentence first, the effects of adjacent words have a greater effect when generating outputs. It would be wiser to read the *whole* sentence and then start to generate a sequence. This is the motivation for the **encoder-decoder model**. It is conventionally divided into a two-stage network.

1. The encoder neural net would convert a sequence into a single latent space representation $z = f(x)$. This latent representation z essentially refers to a feature (vector) representation, which is able to capture the underlying semantic information of the input that is useful for predicting the output.
2. The decoder neural net would decode this feature vector, called the **context vector**, into a sequence of the desired output $y = g(z)$ by using it as the initial hidden state. It uses the previous output as the next input for decoding.

Note that the encoder and decoder are two completely separate neural networks with their own parameters. This is important, since the fact that these are two completely separate networks allows us to work in different "paradigms" within either the feature or target space. For example, if we want to perform machine translation from English to Spanish, our encoder RNN parameters have been tuned to the English syntax and language, while the decoder RNN parameters are tuned to the Spanish language. Since we are modeling different languages, it makes sense to have different sequence models for each one.

We will talk about a specific type of encoder-decoder model called **seq2seq**, which maps sequences to sequences using RNN encoders and decoders. Conventionally, the hidden nodes of the encoder are denoted with $\mathbf{h}$, and those of the decoder are denoted with $\mathbf{s}$.

1. For the encoder, we take in the inputs $\mathbf{x}_t$ and generate the hidden states as

$$\mathbf{h}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1}) = \mathbf{W}_e \mathbf{h}_{t-1} + \mathbf{U}_e \mathbf{x}_t + \mathbf{b}_e$$

   In general, the encoder transforms the hidden states at all time steps into a context variable through the composition of functions q

$$\mathbf{C} = q(\mathbf{h}_1, \mathbf{h}_2, \dots, \mathbf{h}_T)$$

   In the figure below, the context variable is just $\mathbf{C} = \mathbf{h}_T$.

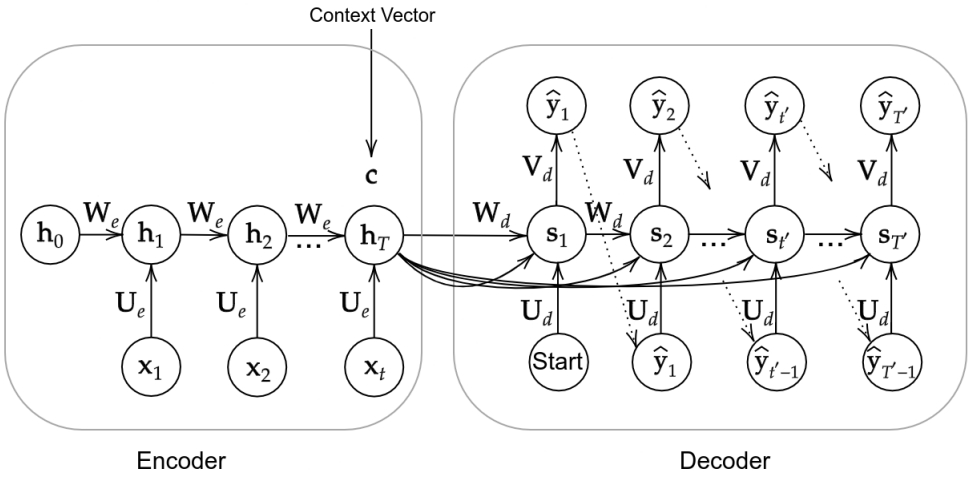
2. Now, given the target output sequence $\hat{\mathbf{y}}_1, \dots, \hat{\mathbf{y}}_{t'+1}$ for each timestep t' (we use t' to differentiate from the input sequence time steps), the decoder assigns a predicted probability to each possible token occurring at step $\hat{\mathbf{y}}_{t'+1}$ conditioned on both the previous tokens $\hat{\mathbf{y}}_1, \dots, \hat{\mathbf{y}}_{t'+1}$ and the context variable $\mathbf{C}$, i.e.

$$\mathbb{P}(\hat{\mathbf{y}}_{t'+1} \mid \hat{\mathbf{y}}_1, \dots, \hat{\mathbf{y}}_{t'+1}, \mathbf{C})$$

Therefore, to decode the subsequent token $\hat{\mathbf{y}}_{t'+1}$, we calculate the hidden state $\mathbf{s}_{t'+1}$ as a gated hidden unit computed by

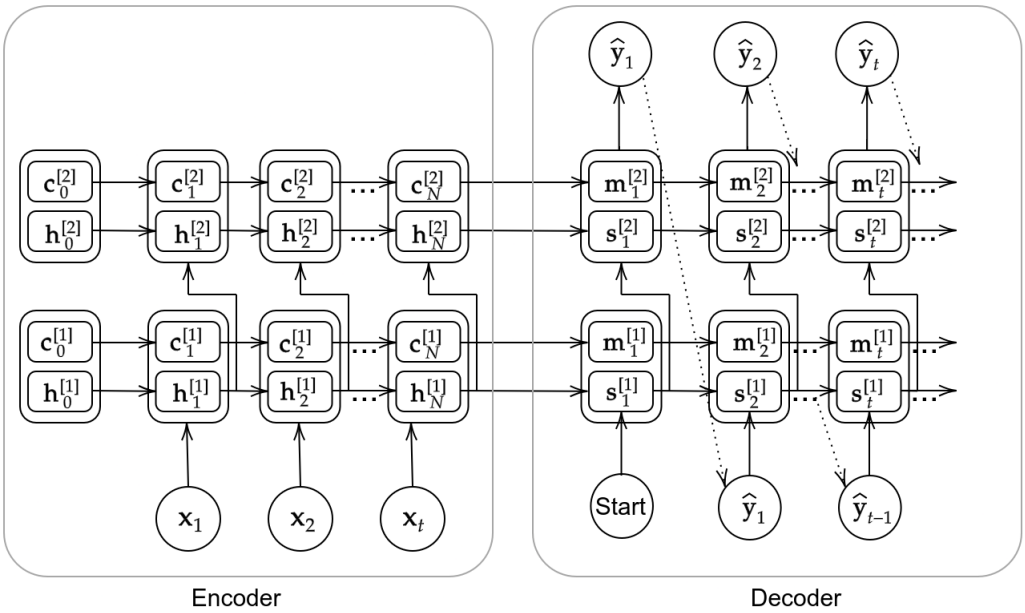
$$\mathbf{s}_{t'+1} = g(\mathbf{s}_{t'}, \hat{\mathbf{y}}_{t'}, \mathbf{C})$$

with the math mentioned here.



Again, note that this encoder-decoder model is comprised of two completely separate deep models with their own parameters, and so it is *not* simply just one long RNN that starts generating outputs only after it takes in all the inputs. Sometimes, the inputs to the decoder may not be shown in diagrams since it is assumed that they are always the previous node's outputs. Furthermore, we can also see that there is no clear-defined first input for the decoder model, since this is the beginning of the sequence. We usually just put some special "start" element in here to denote the beginning of the output.

Here is a diagram for a encoder-decoder model for a 2-layer LSTM which is the standard for practical use, which encodes the sentence meaning in the vectors $\mathbf{c}_t^{[2]}, \mathbf{h}_t^{[2]}, \mathbf{c}_t^{[1]}, \mathbf{h}_t^{[1]}$. In practice, high performing RNNs are usually multilayer (almost alway greater than 1, but diminishing performance returns as number of layers increases), but are not as deep as convolutional or feed forward networks.



Again, to train this model, we do the same backpropagation algorithm on a normalized loss function with teacher forcing over a parallel dataset. What is nice about the encoder-decoder seq2seq is that it can be

completely implemented end-to-end, so we can backpropagate through the entire decoder and encoder to train the both models simultaneously.

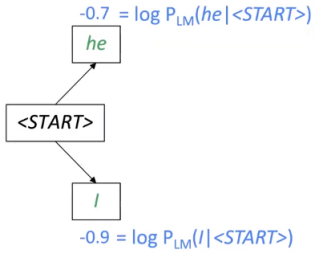
5.2.1 Decoding Schemes

Note that every sequential output of the decoder takes the output of the final layer hidden cell, multiplies it by some matrix, and finally invokes some activation function on it. Consider a classification problem where we have V classes, with a linear map mapping to $\mathbb{R}^V$, followed by a softmax activation. It seems most natural to choose the class that has the maximum probability from the softmax, but this greedy algorithmic approach may not be ideal since we may be giving up long term benefits for short term ones. What we really want to do find the sequence y that maximizes

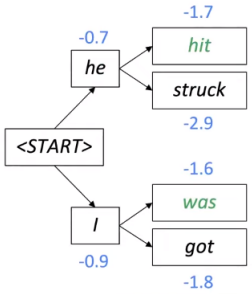
$$\mathbb{P}(y \mid x) = \prod_{t=1}^{T} \mathbb{P}(y_t \mid y_1, \dots, y_{t-1}, x)$$

Clearly, computing the joint probability distribution over all sequences is too expensive, so we can do **beam search decoding**. The main idea is that on each step of the decoder, we keep track of the k (in practice around 5 to 10) most probable partial outputs. For example in the case of machine translation, given a beam size of $k = 2$, we can keep track of the (log) probabilities of the sequences and only keep track of the top 2.

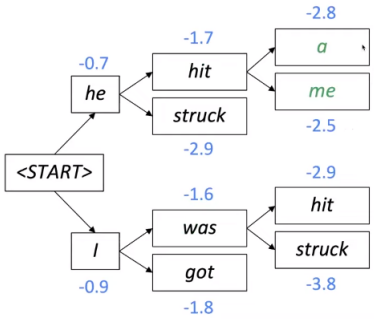
1. Given the START token, say that the k most probable next words were “he" (-0.7) and “I" (-0.9).



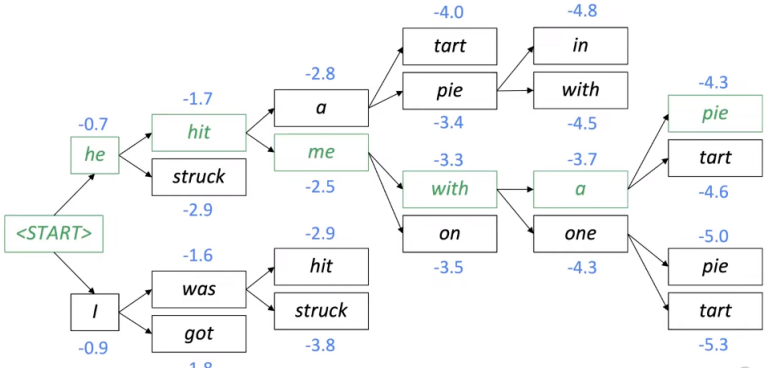
2. Now we look at the two most likely next words for each of “he" and “I" and out of the four possibilities, we compute the two most likely ones, which is “he hit" (-1.7) and “I was" (-1.6).



3. We keep track of “he hit" and “I was" and find the two most likely next words for each, leading us to another four possibilities. We compute the two most likely ones, which is “he hit me" (-2.5) and “he hit a" (-2.8).



4. We keep repeating this.



One more condition to mention is when to stop generating words. In greedy decoding, we usually decode until the model produces an END token. In beam search decoding, different hypotheses may produce END tokens on different timestamps, and so every time we have a complete hypothesis, we can place it aside and continue exploring other hypotheses via beam search. We can continue beam searching until we reach some predetermined cutoff timestep T or we have at least n completed hypotheses (where n is also some predetermined cutoff). We have a slight problem that longer hypotheses will have lower log probabilities, so we can choose the best output sequence by taking the average log probabilities (which corresponds to the geometric mean of the probabilities).

$$\text{score}(y_{t+1}, \dots, y_T) = \frac{1}{T-t} \log \mathbb{P}_{LM}(y_1, \dots, y_t \mid x) = \frac{1}{T-t} \sum_{k=t+1}^{T} \log \mathbb{P}_{LM}(y_k \mid y_{t+1}, \dots, y_{k-1}, x)$$

5.3 More Flexible Models

By combining these neural nets, we can essentially create image captioning (with a CNN encoder and RNN decoder) and image generation (RNN encoder and CNN decoder).

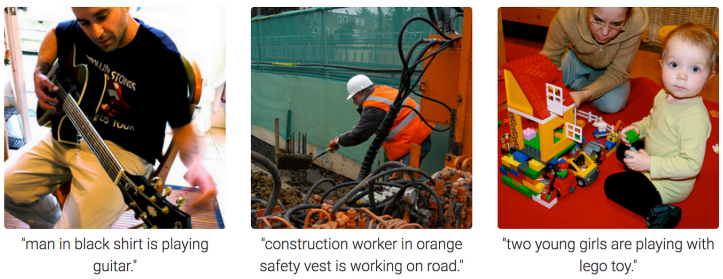


Figure 8: Image captioning on various image prompts.

6 Boltzmann Machines

6.1 Graphical Models

Graphical models allow a nice way to represent complex probability distributions with some dependence relationship.

6.1.1 Directed Graphical Models

Definition 6.1 (Directed Probability Graph)

A **probability graph** is a directed acyclic graph of M nodes representing a joint probability distribution of M scalar random variables. An edge pointing $A \to B$ means that the B is conditionally dependent on A, and that there is a very clear casual relationship coming from A to B.

graphics need to be added

The **parents** of a node x_i is denoted pa_i, and the entire joint distribution can be broken up as such:

$$p(\mathbf{x}) = \prod_{m=1}^{M} p(x_m \mid x_{\text{pa}_m})$$

Example 6.1 (Relay Race)

TBD

Bayesian modelling with hierarchical priors.

Definition 6.2 (Ancestral Sampling)

We can sample from the joint distribution by sequentially sampling starting from the parents to the final children, and discarding the ones (marginalizing) that we don't wish to sample.

Example 6.2 ()

We first provide some motivation from a computational complexity perspective. Given a joint distribution of 2 random variables $\mathbf{x}_1, \mathbf{x}_2$, say which are multinomial with K classes, their joint distribution $p(\mathbf{x}_1, \mathbf{x}_2)$ is captured by $K^2 - 1$ parameters. For a general M random variables, then we have to keep a total of $K^M - 1$ parameters, and this increases exponentially.

TBD

By building a directed graph with say r maximum number of variables appearing on either side of the conditioning bar in a single probability distribution, then the computational complexity scales as $O(K^r)$, which may save a lot of effort if $r << M$.

Extending upon this example, we can see that we want to balance two things:

1. Fully conncted graphs have completely general distributions and have $O(K^M - 1)$ number of parameters (too complex).
2. If there are no links, the joint distribution fully factorizes into the product of its marginals and has $M(K-1)$ parameters (too simple) .

Graphs that have an intermediate level of connectivity allow for more general distributions compared to the fully factorized one, while requiring fewer parameters than the general joint distribution. One model that balances this out is the hidden markov model.

Example 6.3 (Chain Graph)

Consider an M-node Markov chain. The marginal distribution $p(\mathbf{x}_1)$ requires $K-1$ parameters, and the remaining conditional distributions $p(\mathbf{x}_i \mid \mathbf{x}_{i-1})$ requires $K(K-1)$ parameters. Therefore, the total number of parameters is

$$K - 1 + (M-1)(K-1)K \in O(MK^2)$$

which scales relatively well, and we have

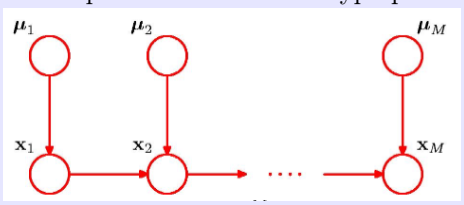
$$p(\{\mathbf{x}_m\}) = p(\mathbf{x}_1) \prod_{m=2}^{M} p(\mathbf{x}_m \mid \mathbf{x}_{m-1})$$

TBD

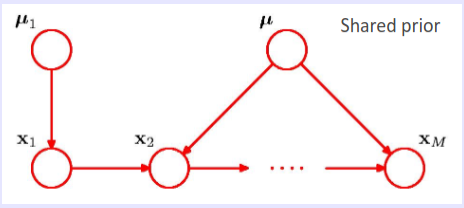
We can turn this same graph into a Bayesian model by introducing priors for the paramters. Therefore, each node requires an additional parent representing the distribution over parameters (e.g. prior can be Dirichlet)

$$p(\{\mathbf{x}_m, \mu_m\}) = p(\mathbf{x}_1 \mid \mu_1)p(\mu_1) \prod_{m=2}^{M} p(\mathbf{x}_m \mid \mathbf{x}_{m-1}, \mu_m)p(\mu_m)$$

with $p(\mu_m) = \text{Dir}(\mu_m \mid \alpha_m)$ for some predetermined fixed hyperparameter α_m.



We could also choose to share a common prior over the parameters, trading flexibility for computational feasibility.



Another way to make more compact representations is through parameterized models. For example, if we have to compute $p(y = 1 \mid \mathbf{x}_1, \dots, \mathbf{x}_M)$, this in general has $O(K^M)$ parameters. However, we can obtain a more parsimonious form by using a logistic function acting on a linear combination of the parent variables

$$p(y = 1 \mid \mathbf{x}_1, \dots, \mathbf{x}_m) = \sigma\left(w_0 + \sum_{i=1}^{M} w_i x_i\right) = \sigma(\mathbf{w}^T \mathbf{x})$$

We can look at an example how this is applied to sampling from high-dimensional Gaussian with **linear Gaussian models**.

Example 6.4 (Multivariate Gaussian)

Consider an arbitrary acyclic graph over D random variables, in which eachnode represents a single continuous Gaussian distribution with its mean given by a linear function of its parents.

$$p(x_i \mid \mathbf{pa}_i) = N\left(x_i \middle| w_{ij}x_j + b_j, v_i\right)$$

Given a multivariate Gaussian, let us try to decompose it into a directed graph. The log of the joint distribution takes form

$$\ln p(\mathbf{x}) = \sum_{i=1}^{D} \ln p(x_i \mid \text{pa}_i) = -\sum_{i=1}^{D} \frac{1}{2v_i}\left(x_i - \sum_{j \in \text{pa}_i} w_{ij}x_j - b_i\right)^2 + \text{const}$$

To compute the mean, we can see that by construction, every x_i is dependent on its ancestors, so

$$x_i = \sum_{j \in \text{pa}_i} w_{ij}x_j + b_i + \sqrt{v_i}\epsilon_i, \quad \epsilon_i \sim N(0,1)$$

so by linearity of expectation, we have

$$\mathbb{E}[x_i] = \sum_{j \in \text{pa}_i} w_{ij}\mathbb{E}[x_j] + b_i$$

So again, we can start at the top of the graph and compute the expectation. To compute covariance, we can obtain the i,jth element of $\mathbf{\Sigma}$ with a recurrence relation:

$$\begin{aligned}
\Sigma_{ij} &= \mathbb{E}[(x_i - \mathbb{E}[x_i])(x_j - \mathbb{E}[x_j])] \\
&= \mathbb{E}\left[(x_i - \mathbb{E}[x_i])\left(\sum_{k \in \text{pa}_j} w_{jk}(x_k - \mathbb{E}[x_k]) + \sqrt{v_i}\epsilon_j\right)\right] \\
&= \sum_{k \in \text{pa}_j} w_{jk}\Sigma_{ik} + I_{ij}v_j
\end{aligned}$$

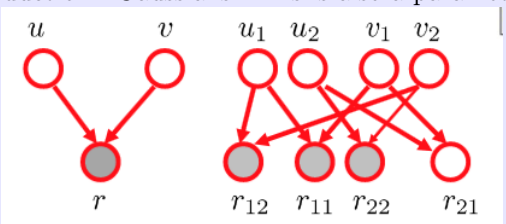
If there were no links in the graphs, then the w_{ij}'s are 0, and so $\mathbb{E}[\mathbf{x}] = [b_1, \dots, b_D]$, making the covariance diagonal.If the graph is fully connected, then the total number of parameters is $D + D(D-1)/2$, which corresponds to a general symmetric covariance matrix.

Example 6.5 (Bilinear Gaussian Model)

Consider the following model

$$\begin{aligned}
u &\sim N(0,1) \\
v &\sim N(0,1) \\
r &\sim N(uv, 1)
\end{aligned}$$

where the mean of r is a product of 2 Gaussians. This is also a parameterized model.



Definition 6.3 (Conditional Independence in Directed Graphs)

We say that a is independent of b given c if

$$p(a \mid b, c) = p(a \mid c)$$

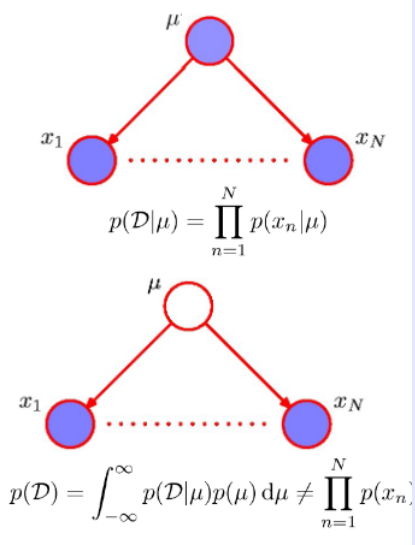
or equivalently,

$$p(a, b \mid c) = p(a \mid b, c)\, p(b \mid c) = p(a \mid c)\, p(b \mid c)$$

Conveniently, we can directly read conditional independence properties of the joint distribution from the graph without any analytical measurements.

Example 6.6 ()

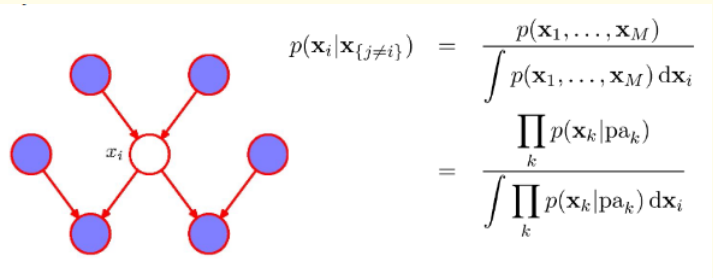
We can demonstrate conditional independence with iid data. Consider the problem of density estimation of some dataset $\mathcal{D} = \{x_i\}$ with some parameterized distribution of μ. As shown below, if we condition on μ and considered the joint over the observed variables, the variables are independent, but if we integrate out μ, the observations are no longer independent.



The example above identifies a node (the parent μ) where, if observed, causes the rest of the nodes to become independent. We can extend on this idea by taking an arbitrary x_i and finding a set of nodes such that if they are observed, then x_i is indepedent from every other node.

Definition 6.4 (Markov Blanket)

The **Markov blanket** of a node is the minimal set of nodes that must be observed to make this node independent of all other nodes. It turns out that the parents, children, and coparents are all in the Markov blanket.



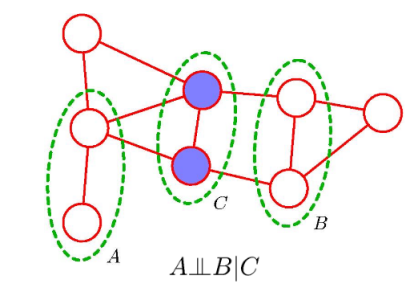
One final interpretation is that we can view directed graphs as **distribution filters**. We take the joint probability distribution, will starts off as fully connected, and the directed graphs "filters" away the edges that are not needed. Therefore, the joint probability distribution $p(\mathbf{x})$ is only allows through the filter if and only if it satisfies the factorization property.

6.1.2 Undirected Graphical Models

As the name implies, undirected models use undirected graphs, which are used to model relationships that go both ways rather than just one. Unlike directed graphs, which are useful for expressing casual relationships between random variables, undirected graphs are useful for expressing soft constraints between random variables.

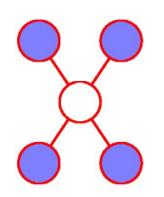
Definition 6.5 (Conditional Independence in Undirected Graphs)

Fortunately, conditional independence is easier compared to directed models. We can say A is conditionally independent to B given C if C blocks all paths between any node in A and any node in B.



Definition 6.6 (Markov Blanket in Undirected Graphs)

The Markov blanket of a node, which is the minimal set of nodes that must be observered to make this node independent of the rest of the nodes, is simply the nodes that are directly connected to that node.



Therefore, the conditional distribution of x_i conditioned on all the variables in the graph is dependent only on the variables in the Markov blanket.

Now, let us talk about how we can actually define a probability distribution with this graph.

Definition 6.7 (Clique)

In an undirected graph, a **clique** is a set of nodes such that there exists a link between all pairs of nodes in that subset. A **maximal clique** is a clique such that it is not possible to include any other nodes in the set without it ceasing it to be a clique.

Given a joint random variable $\mathbf{x}$ represented by an undirected graph, the joint distribution is given by the product of non-negative potential functions over the maximal cliques

$$p(\mathbf{x}) = \frac{1}{Z} \prod_C \phi_C(x_C)$$

where

$$Z = \int p(\mathbf{x})\, d\mathbf{x}$$

is the normalizing constant, called the **partition function**. That is, each x_C is a maximal clique and ϕ_C is the nonnegative potential function of that clique.

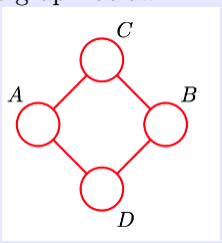
This assignment looks pretty arbitrary. How do we know that any arbitrary joint distribution of $\mathbf{x}$, which has a undirected graphical representation, can be represented as the product of a bunch of functions over the maximum cliques? Fortunately, there is a mathematical result that proves this.

Theorem 6.1 (Hammersley-Clifford)

The joint probability distribution of any undirected graph can be written as the product of potential functions on the maximal cliques of the graph. Furthermore, for any factorization of these potential functions, there exists an undirected graph for which is the joint.

Example 6.7 ()

For example, the joint distribution of the graph below



factorizes into

$$p(A,B,C,D) = \frac{1}{Z}\phi(A,C)\,\phi(C,B)\,\phi(B,D)\,\phi(A,D)$$

Note that each potential function ϕ is a mapping from the joint configuration of random variables in a clique to non-negative real numbers. The choice of potential functions is not restricted to having specific probabilistic interpretations, but since they must be nonnegative, we can just represent them as an exponential. The negative sign is not needed, but is a remnant of physics notation.

$$p(\mathbf{x}) = \frac{1}{Z}\prod_C \phi_C(x_C) = \frac{1}{Z}\exp\left\{-\sum_C E(x_C)\right\} = \frac{1}{Z}\underbrace{\exp\{-E(\mathbf{x})\}}_{\substack{\text{Boltzmann}\\ \text{distribution}}}$$

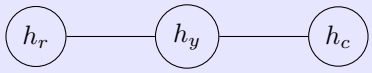
Any distribution that can be represented as the form above is called a **Boltzmann distribution**. So far, all we stated is that the joint probability distribution can be expressed as the product of a bunch of potential functions, but besides the fact that it is nonnegative, there is no probabilistic interpretation of these potentials (or equivalently, the energy functions). While this does give us greater flexibility in choosing potential functions, we must be careful in choosing them (e.g. choosing something like x^2 may cause the integral to diverge, making the joint not well-defined).

Clearly, these potential functions over the cliques should express which configuration of the local variables are preferred to others. It should assign higher values to configurations that are deemed (either by assumption or through training data) to be more probable. That is, each potential is like an "expert" that provides some opinion (the value) on a configuration, and the product of the values of all the potential represents the total opinion of all the experts. Therefore, global configurations with relatively high probabilities are those that find a good balance in satisfying the (possibly conflicting) influences of the clique potentials.

Example 6.8 (Transmission of Colds)

Say that you want to model a distribution over three binary variables: whether you or not you, your coworker, and your roommate is sick (0 represents sick and 1 represents healthy). Then, you can make simplifying assumptions that your roommate and your coworker do not know each other, so it is very unlikely that one of thme will give the other an infection such as a cold directly. Therefore, we can model the indirect transmission of a cold from your coworker to your roommate by modeling

the transmission of the cold from your coworker to you and then you to your roommate. Therefore, we have a model of form



One max clique contains h_y and h_c. The factor for this clique can be defined by a table and might have values resembling these.

	$h_y = 0$	$h_y = 1$
$h_c = 0$	2	1
$h_c = 1$	1	10

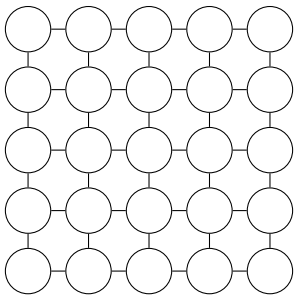
Table 1: States and Values of h_y and h_c

This table completely describes the potential function of this clique. Both of you are usually healthy, so the state $(1, 1)$ gets the maximum value of 1. If one of you are sick, then it is likely that the other is sick as well, so we have a value of 2 for $(0, 0)$. Finally, it is most unlikely that one of you is sick and the other healthy, which has a value of 1.

6.2 Boltzmann Machines

Now that we've learned about graphical models, let's put them to use. We have some unknown joint distribution $\mathbf{x}$, and we want to represent it in a graph such that it is not too computationally hard to calculate probabilities and sample from them, but at the same time not so simple such that it doesn't richly capture a broad family of probability distributions. One architecture is to use **Markov Random Fields**, which represent these joint distributions with undirected graphs satisfying the Markov properties.

The Hammersley-Clifford theorem states that the joint PDF of any MRF can be written a a Boltzmann distribution. For now, we will limit outselves to **parwise MRFs**, which only capture dependencies between cliques of maximum size 2. For example, a MRF can be represented with the graph $G(V, E)$ below.



Definition 6.8 (Bernoulli Pairwise Markov Random Fields)

MRFs with binary variables are sometimes **Ising models** in statistical mechanics, and **Boltzmann machines** in machine learning. By Hammersley-Clifford, we don't even need to specify the individual functions over the maximal cliques, and rather we can just specify the energy function $E(\mathbf{x})$ of the Boltzmann distribution that the MRF encodes. We define it to capture the interactions between random variables x_i up to order 2.

$$p_{\boldsymbol{\theta}}(\mathbf{x}) = \frac{1}{Z} \exp\left(\sum_{ij \in E} x_i x_j \theta_{ij} + \sum_{i \in V} x_i \theta_i \right)$$

Now let's check its conditional distribution.

$$\begin{aligned} p(x_k = 1 \mid \mathbf{x}_{-k}) &= \frac{p(x_k = 1, \mathbf{x}_{-k})}{p(\mathbf{x}_{-k})} \\ &= \frac{p(x_k = 1, \mathbf{x}_{-k})}{p(x_k = 0, \mathbf{x}_{-k}) + p(x_k = 1, \mathbf{x}_{-k})} \\ &= \frac{\exp\left(\sum_{kj \in E} x_j \theta_{kj} + x_k \theta_k \right)}{\exp(0) + \exp\left(\sum_{kj \in E} x_j \theta_{kj} + x_k \theta_k \right)} \\ &= \sigma\left\{ -\theta_k x_k - \sum_{kj \in E} x_j \theta_{kj} \right\} \end{aligned}$$

where the penultimate step comes from evaluating

$$\begin{aligned} p(x_k = 1, \mathbf{x}_{-k}) &= \frac{1}{Z(\boldsymbol{\theta})} \exp\left(\sum_{ij \in E, k \neq i,j} x_i x_j \theta_{ij} + \sum_{ij \in E, k = i,j} x_i x_j \theta_{ij} + \sum_{i \in V, i \neq k} x_i \theta_i + x_k \theta_k \right) \\ &= \frac{1}{Z(\boldsymbol{\theta})} \exp\left(\sum_{ij \in E, k \neq i,j} x_i x_j \theta_{ij} + \sum_{kj \in E} x_j \theta_{kj} + \sum_{i \in V, i \neq k} x_i \theta_i + \theta_k \right) \\ p(x_k = 0, \mathbf{x}_{-k}) &= \frac{1}{Z(\boldsymbol{\theta})} \exp\left(\sum_{ij \in E, k \neq i,j} x_i x_j \theta_{ij} + \sum_{i \in V, i \neq k} x_i \theta_i \right) \end{aligned}$$

and canceling out like terms in the numerator and denominator. This tells us that MRFs are related to logistic function.

We have given our first example of a Boltzmann machine. Let's generalize this a little bit by removing the restriction that there can only be pairwise connections. Then, we can model the second order interactions with the slightly more generalized energy function

$$E(\mathbf{x}) = -\mathbf{x}^T \mathbf{U} \mathbf{x} - \mathbf{b}^T \mathbf{x}$$

Now this slightly expands the coverage of probability distributions given our model, and we can see that this allows us to model Gaussian distributions.

Example 6.9 (Gaussian Markov Random Fields)

If we assume that $p_{\boldsymbol{\theta}}(\mathbf{x})$ folows a multivariate Gaussian distribution, we have

$$p(\mathbf{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{Z} \exp\left(-\frac{1}{2} (\mathbf{x} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu}) \right)$$

Since the Gaussian distribution reprsents at most second-order relationships, it automatically encodes a pairwise MRF. Therefore, we can rewrite

$$p(\mathbf{x}) = \frac{1}{Z} \exp\left(-\frac{1}{2} \mathbf{x}^T J \mathbf{x} + \mathbf{g}^T \mathbf{x} \right)$$

where $J = \boldsymbol{\Sigma}^{-1}$ and $\boldsymbol{\mu} = J^{-1}\mathbf{g}$.

Let's review what we had so far. There is a random vector $\mathbf{x}$ for which we would like to model the probability distribution of.

What we can do is model the dependencies between these random elements with linear parameters $\mathbf{W}$ and $\mathbf{b}$, which essentially gives us a Markov Random Field.

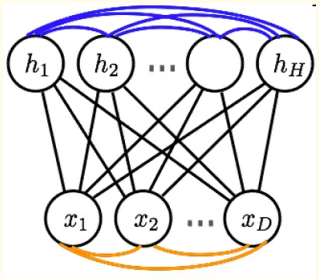
However, this is still quite a limited model. For one, due to the linearity of the weight matrix, it always turns out that the probability of $x_k = 1$ is always given by a linear model (logistic regression) from the values of the other units. This family of distributions parameterized by $\boldsymbol{\theta} = \{\mathbf{W}, \mathbf{b}\}$ may not be broad enough to capture the true $p(\mathbf{x})$. Therefore, we can add latent variables that can act similarly to hidden uits in a MLP and model higher-order interactions among the visible units. Just as the addition of hidden units to convert logistic regression into MLP results in the MLP being a universal approximator of functions, a Boltzmann machine with hidden units is not longer limited to modeling linear relationships between variables. Instead, the Boltzmann machine becomes a universal approximator of probability mass functions over discrete random variables.

Definition 6.9 (Boltzmann Machine)

The original **Boltzmann machine** has the energy function

$$E(\mathbf{v}, \mathbf{h}) = -\mathbf{v}^T\mathbf{R}\mathbf{v} - \mathbf{v}^T\mathbf{W}\mathbf{h} - \mathbf{h}^T\mathbf{S}\mathbf{h} - \mathbf{b}^T\mathbf{v} - \mathbf{c}^T\mathbf{h}$$

It can represent the undirected graph that has connections within the $\mathbf{x}$, within the $\mathbf{h}$, and between the $\mathbf{x}$ and $\mathbf{h}$.



Therefore, by adding latent variables and connecting everything together, this gives us a very flexible model that can capture a lot of distribtions.

6.2.1 Restricted Boltzmann Machines

Definition 6.10 (Restricted Boltzmann Machine)

Now, if we put a restriction saying that there cannot be any intra-connections in the $\mathbf{x}$ and $\mathbf{h}$, then we get the **restricted Boltzmann machine**, which has a slightly more resticted form of the energy function than the general BM. The probability distributions that it can model has a graph that looks like

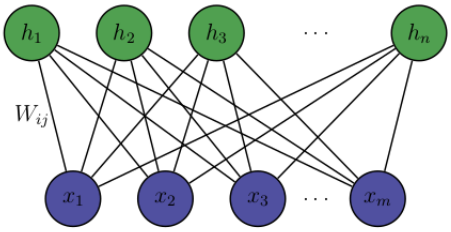


Figure 1: Graphical description of Restricted Boltzmann Machine

with connections only allowed between x_i's and h_j's, known as a **bipartite graph**, implying that the maximum clique length is 2. This model allows the elements of $\mathbf{x}$ to be dependent, but this architecture allows for *conditional independence*, and not just for $\mathbf{x}$ given $\mathbf{h}$, but also $\mathbf{h}$ given $\mathbf{x}$.

Therefore, we already have the extremely nice property that

$$p(\mathbf{x} \mid \mathbf{h}) = \prod_{k=1}^{D} p(x_k \mid \mathbf{h})$$
$$p(\mathbf{h} \mid \mathbf{x}) = \prod_{j=1}^{F} p(h_j \mid \mathbf{x})$$

The fact that we can calculate $p(\mathbf{h} \mid \mathbf{x})$ means that inferring the distribution over the hidden variables is easy. Keep in mind that there are three architectures we've talked about:

1. Markov Random Fields, which model just the original $\mathbf{x}$.
2. Restricted Boltzmann machines, which models $\mathbf{x}, \mathbf{h}$ and not allowing intra-connections.
3. Boltzmann machines, which models $\mathbf{x}, \mathbf{h}$. Boltzmann machines without latent variables are just MRFs.

Definition 6.11 (Bernoulli-Bernoulli RBM)

For now, let us assume that we are trying to estimate the distribution of a Bernoulli random vector $\mathbf{x} \in \{0,1\}^D$ with Bernoulli latent variables $\mathbf{h} \in \{0,1\}^F$. Then, the energy of the joint configuration is

$$E(\mathbf{v}, \mathbf{h}; \boldsymbol{\theta}) = -\sum_{ij} W_{ij} v_i h_j - \sum_i b_i v_i - \sum_j a_j h_j = -\mathbf{v}^T \mathbf{W} \mathbf{h} - \mathbf{b}^T \mathbf{v} - \mathbf{a}^T \mathbf{h}$$

where $\boldsymbol{\theta} = \{\mathbf{W}, \mathbf{a}, \mathbf{b}\}$ are the model parameters. So we have

$$p_{\boldsymbol{\theta}}(\mathbf{v}, \mathbf{h}) = \frac{1}{Z} \exp\big(-E(\mathbf{v}, \mathbf{h}; \boldsymbol{\theta})\big) = \frac{1}{Z} \prod_{ij} e^{W_i j v_i h_j} \prod_i e^{b_i v_i} \prod_j e^{a_j h_j}$$
$$Z = \sum_{\mathbf{h}, \mathbf{v}} \exp\big(-E(\mathbf{v}, \mathbf{h}; \boldsymbol{\theta})\big)$$

where we can think of the $\exp(\mathbf{h}^T \mathbf{W} \mathbf{x})$ as encoding the cliques of length 2 and the others as cliques of length 1.

Let's get some calculations out of the way.

Lemma 6.1 (Conditional Distributions)

For the Bernoulli RBM, we have

$$p(h_j = 1 \mid \mathbf{x}) = \sigma(b_j + W_{j,:} \mathbf{x})$$
$$p(x_k = 1 \mid \mathbf{h}) = \sigma(c_k + \mathbf{h}^T \mathbf{W}_{:,k})$$

Proof.

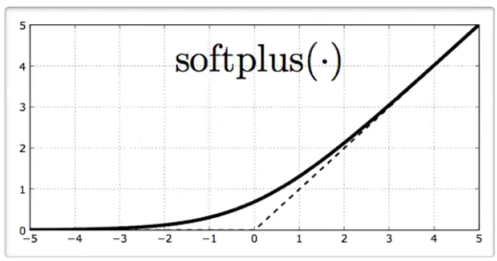
Just use the definition of conditional probability and substitute the result below in the denominator. The terms will cancel out.

Lemma 6.2 (Free Energy)

For the Bernoulli RBM, we want to compute the marginal $p(\mathbf{x})$ as

$$\begin{aligned} p(\mathbf{x}) &= \frac{\exp(-F(\mathbf{x}))}{Z} \\ &= \frac{1}{Z}\exp\left(\mathbf{c}^T\mathbf{x} + \sum_{j=1}^{H}\log\left(1+\exp(b_j+\mathbf{W}_{j,:}\mathbf{x})\right)\right) \\ &= \frac{1}{Z}\exp\left(\mathbf{c}^T\mathbf{x} + \sum_{j=1}^{H}\text{softplus}(b_j+W_{j,:}\mathbf{x})\right) \end{aligned}$$

where F is called the **free energy**. Therefore, $p(\mathbf{x})$ is calculated by taking the product of these terms, which is why it's known as a **product of experts model**.



Proof.

We have

$$\begin{aligned} p(\mathbf{x}) &= \sum_{\mathbf{h}\in\{0,1\}^H}\exp\left(\mathbf{h}\mathbf{W}\mathbf{x} + \mathbf{c}^T\mathbf{x} + \mathbf{b}^T\mathbf{h}\right)/Z \\ &= \exp(\mathbf{c}^T\mathbf{x})\sum_{h_1=0,1}\cdots\sum_{h_H=0,1}\exp\left(\sum_j h_j\mathbf{W}_{j,:}\mathbf{x} + b_jh_j\right)/Z \\ &= \exp(\mathbf{c}^T\mathbf{x})\left(\sum_{h_1=0,1}\exp(h_1\mathbf{W}_{1,:}\mathbf{x}+b_1h_1)\right)\cdots\left(\sum_{h_H=0,1}\exp(h_H\mathbf{W}_{H,:}\mathbf{x}+b_Hh_H)\right)/Z \\ &= \exp(\mathbf{c}^T\mathbf{x})\left(1+\exp(b_1+\mathbf{W}_{1,:}\mathbf{x})\right)\cdots\left(1+\exp(b_H+\mathbf{W}_{H,:}\mathbf{x})\right)/Z \\ &= \exp(\mathbf{c}^T\mathbf{x})\exp\left\{\log\left(1+\exp(b_1+\mathbf{W}_{1,:}\mathbf{x})\right)\right\}\cdots\exp\left\{\log\left(1+\exp(b_H+\mathbf{W}_{H,:}\mathbf{x})\right)\right\}/Z \\ &= \frac{1}{Z}\exp\left(\mathbf{c}^T\mathbf{x} + \sum_{j=1}^{H}\log\left(1+\exp(b_j+\mathbf{W}_{j,:}\mathbf{x})\right)\right) \end{aligned}$$

Now that we've done this, we can finally get to training the model. Now, essentially this is density estimation problem given dataset $\mathcal{D} = \{\mathbf{x}^{(t)}\}$ of iid random variables, we want to maximize the likelihood of $p_{\boldsymbol{\theta}}$, which is really just equivalent to optimizing $E_{\boldsymbol{\theta}}$. So, let's take the average negative log-likelihood and take the derivative of it

$$\frac{\partial}{\partial\boldsymbol{\theta}}\frac{1}{T}\sum_t -\log p_{\boldsymbol{\theta}}(\mathbf{x}^{(t)}) = \frac{1}{T}\sum_t -\log p_{\boldsymbol{\theta}}(\mathbf{x}^{(t)})$$

There's a lot of computation to do here, so let's focus on one sample $\mathbf{x}^{(t)}$ and claim that the gradient ultimately ends up as the following.

Theorem 6.2 ()

It turns out that

$$\begin{aligned}\frac{\partial}{\partial \boldsymbol{\theta}} - \log p(\mathbf{x}^{(t)}) &= \sum_{\mathbf{h}} p(\mathbf{h} \mid \mathbf{x}^{(t)}) \frac{\partial E(\mathbf{x}^{(t)}, \mathbf{h})}{\partial \boldsymbol{\theta}} - \sum_{\mathbf{x},\mathbf{h}} p(\mathbf{x}, \mathbf{h}) \frac{\partial E(\mathbf{x}, \mathbf{h})}{\partial \boldsymbol{\theta}} \\ &= \mathbb{E}_{\mathbf{h}}\left[\frac{\partial E(\mathbf{x}^{(t)}, \mathbf{h})}{\partial \boldsymbol{\theta}} \;\middle|\; \mathbf{x}^{(t)}\right] - \mathbb{E}_{\mathbf{x},\mathbf{h}}\left[\frac{\partial E(\mathbf{x}, \mathbf{h})}{\partial \boldsymbol{\theta}}\right]\end{aligned}$$

The derivative of E is easy since we already know the bilinear form by construction. In the left term, we are taking the expectation w.r.t. $p(\mathbf{h} \mid \mathbf{x}^{(t)})$, which we can factorize out due to conditional independence, so this is easy. However, the right term requires us to integrate over the joint $p(\mathbf{x}, \mathbf{h})$, which is intractable, and so we just approximate this with a Monte Carlo sample.

Proof.

As a lemma, we first see that $\ln(Z) = \ln\big(\sum_{\mathbf{x},\mathbf{h}} \exp\big(-E(\mathbf{x}, \mathbf{h})\big)\big)$, and so

$$\frac{\partial \ln(Z)}{\partial \boldsymbol{\theta}} = -\frac{1}{Z} \sum_{\mathbf{x},\mathbf{h}} \exp\big(-E(\mathbf{x}, \mathbf{h})\big) \frac{E(\mathbf{x}, \mathbf{h})}{\partial \boldsymbol{\theta}} = -\sum_{\mathbf{x},\mathbf{h}} p(\mathbf{x}, \mathbf{h}) \frac{\partial E(\mathbf{x}, \mathbf{h})}{\partial \boldsymbol{\theta}}$$

We have

$$-\ln p(\mathbf{x}) = -\ln\left\{\sum_{\mathbf{h}} \exp\big(-E(\mathbf{x}, \mathbf{h})\big)\right\} + \ln(Z)$$

and so we can apply chain rule and multiply both numerator and denominator by $1/Z$ to get

$$\begin{aligned}-\frac{\partial}{\partial \boldsymbol{\theta}} \ln p(\mathbf{x}) &= \frac{\sum_{\mathbf{h}} \exp\big(-E(\mathbf{x}, \mathbf{h})\big) \frac{\partial E(\mathbf{x}, \mathbf{h})}{\partial \boldsymbol{\theta}} / Z}{\sum_{\mathbf{h}} \exp\big(-E(\mathbf{x}, \mathbf{h})\big)/Z} + \frac{\partial \ln(Z)}{\partial \boldsymbol{\theta}} \\ &= \frac{\sum_{\mathbf{h}} p(\mathbf{x}, \mathbf{h}) \frac{\partial E(\mathbf{x}, \mathbf{h})}{\partial \boldsymbol{\theta}}}{p(\mathbf{x})} + \frac{\partial \ln(Z)}{\partial \boldsymbol{\theta}} \\ &= \sum_{\mathbf{h}} p(\mathbf{h} \mid \mathbf{x}) \frac{\partial E(\mathbf{x}, \mathbf{h})}{\partial \boldsymbol{\theta}} - \sum_{\mathbf{x},\mathbf{h}} p(\mathbf{x}, \mathbf{h}) \frac{\partial E(\mathbf{x}, \mathbf{h})}{\partial \boldsymbol{\theta}}\end{aligned}$$

So to calculate the second expectation, we can use a Gibbs sampler to do some numerical integration, but before we do that, let's just find the partial of E, which should be simple.

$$\frac{\partial E(\mathbf{x}, \mathbf{h})}{\partial W_{jk}} = \frac{\partial}{\partial W_{jk}}\left(-\sum_{jk} W_{jk} h_j x_k - \sum_k c_k x_k - \sum_j b_j h_j\right) = h_j x_k$$

and so

$$\mathbb{E}_{\mathbf{h}}\left[\frac{\partial E(\mathbf{x}, \mathbf{h})}{\partial W_{jk}} \middle| \mathbf{x}\right] = \mathbb{E}_{\mathbf{h}}[-h_j x_k \mid \mathbf{x}] = \sum_{h_j = 0,1} -h_j x_k \, p(h_j \mid \mathbf{x}) = -x_k p(h_j = 1 \mid \mathbf{x})$$

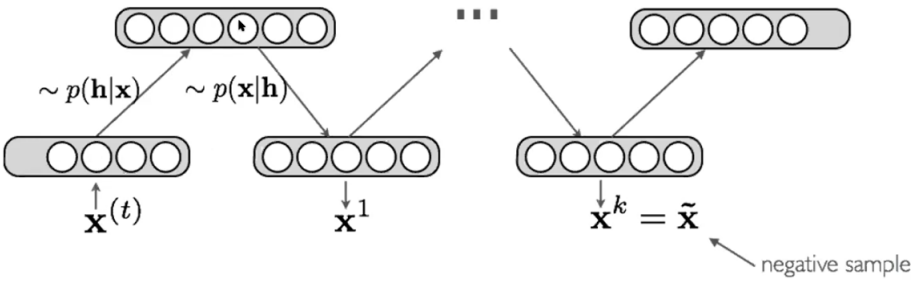
where the final term is a sigmoid. Hence, we have

$$\mathbb{E}_{\mathbf{h}}[\nabla_{\mathbf{W}} E(\mathbf{w}, \mathbf{h}) \mid \mathbf{x}] = -\mathbf{h}(\mathbf{x})\mathbf{x}^T, \text{ where } \mathbf{h}(\mathbf{x}) \coloneqq \begin{pmatrix} p(h_1 = 1 \mid \mathbf{x}) \\ \vdots \\ p(h_H = 1 \mid \mathbf{x}) \end{pmatrix} = \sigma(\mathbf{b} + \mathbf{W}\mathbf{x})$$

Now we can substitute what we solved into the second expectation, but again this is infeasible to calculate

$$\mathbb{E}_{\mathbf{x},\mathbf{h}}\left[\frac{\partial E(\mathbf{x},\mathbf{h})}{\partial \boldsymbol{\theta}}\right] = \sum_{\mathbf{x},\mathbf{h}} \mathbf{h}(\mathbf{x})\mathbf{x}^T p(\mathbf{x},\mathbf{h})$$

The way we do this is through **contrastive divergence**, which estimates the expectation through Gibbs sampling. Since we know $p(\mathbf{x} \mid \mathbf{h})$ and $p(\mathbf{h} \mid \mathbf{x})$ easily, we can start sampling the chain for some predetermined K steps (actually $2K$ since we are sampling the x and h back and forth), and whatever $\bar{\mathbf{x}}$ you sample at the end is your estimate. So, once you should update your gradient, you start at the sample $\mathbf{x}^{(t)}$, run Gibbs for k steps, and use that to estimate your gradient, and then move onto the next sample. We can tweak this procedure, such as **persistent CD**, where instead of initializing the chain to $\mathbf{x}^{(t)}$, we can initialize the chain to the negative sample of the last iteration.



Therefore, for updating $\mathbf{W}$, we get the following

$$\begin{aligned}
\mathbf{W} &= \mathbf{W} - \alpha\big(\nabla_{\mathbf{W}}(-\log p(\mathbf{x}^{(t)}))\big) \\
&= \mathbf{W} - \alpha\big(\mathbb{E}_{\mathbf{h}}[\nabla_{\mathbf{W}} E(\mathbf{x}^{(t)},\mathbf{h}) \mid \mathbf{x}^{(t)}] - \mathbb{E}_{\mathbf{x},\mathbf{h}}[\nabla_{\mathbf{W}} E(\mathbf{x},\mathbf{h})]\big) \\
&= \mathbf{W} - \alpha\big(\mathbb{E}_{\mathbf{h}}[\nabla_{\mathbf{W}} E(\mathbf{x}^{(t)},\mathbf{h}) \mid \mathbf{x}^{(t)}] - \mathbb{E}_{\mathbf{h}}[\nabla_{\mathbf{W}} E(\bar{\mathbf{x}},\mathbf{h}) \mid \bar{\mathbf{x}}]\big) \\
&= \mathbf{W} + \alpha\big(\mathbf{h}(\mathbf{x}^{(t)})(\mathbf{x}^{(t)})^T - \mathbf{h}(\bar{\mathbf{x}})\bar{\mathbf{x}}^T\big)
\end{aligned}$$

and doing this over all three parameters leads to

$$\begin{aligned}
\mathbf{W} &\leftarrow \mathbf{W} + \alpha\big(\mathbf{h}(\mathbf{x}^{(t)})(\mathbf{x}^{(t)})^T - \mathbf{h}(\bar{\mathbf{x}})\bar{\mathbf{x}}^T\big) \\
\mathbf{b} &\leftarrow \mathbf{b} + \alpha\big(\mathbf{h}(\mathbf{x}^{(t)}) - \mathbf{h}(\bar{\mathbf{x}})\big) \\
\mathbf{c} &=\leftarrow \mathbf{c} + \alpha\big(\mathbf{x}^{(t)} - \hat{\mathbf{x}}\big)
\end{aligned}$$

Therefore, contrastive divergence with k iterations gives us the **CD-k algorithm**. In general, the bigger k is, the less biased the estimate of the gradient will be, and in practice $k = 1$ works well for learning good features. The reason this is called contrastive divergence is that in the gradient update step, we have a positive sample and a negative sample that both approximates the expected gradient, which constrasts to each other.

6.2.2 Gaussian Bernoulli RBMs

Now we can talk about Gaussian Bernoulli RBMs.

Definition 6.12 (Gaussian-Bernoulli RBM)

If we assume that $\mathbf{v}$ is a real-valued (unbounded) input that follows a Gaussian distribution (with $\mathbf{h}$ still Bernoulli), then we can add a quadratic term to the energy function

$$E(\mathbf{x},\mathbf{h}) = -\mathbf{h}^T\mathbf{W}\mathbf{x} - \mathbf{c}^T\mathbf{x} - \mathbf{b}^T\mathbf{h} - \frac{1}{2}\mathbf{x}^T\mathbf{x} \tag{130}$$

In this case, $p(\mathbf{x} \mid \mathbf{h})$ becomes a Gaussian distribution $N(\mathbf{c}+\mathbf{W}^T\mathbf{h}, \mathbf{I})$. The training process is slightly harder for this, so what we usually do is normalize the training set by subtracting the mean off each input and dividing the input by the training set standard deviation to get

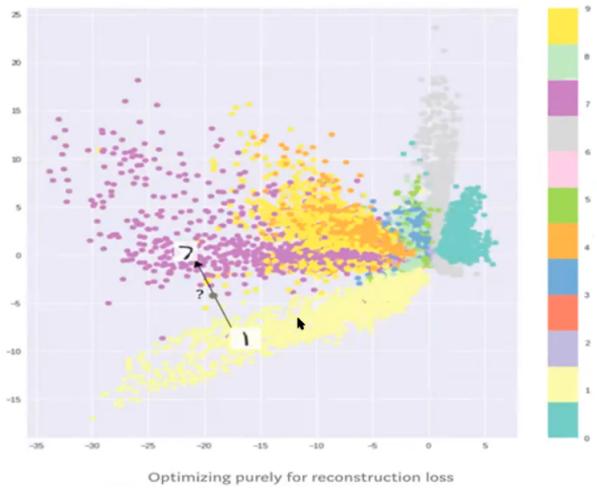
$$E(\mathbf{v}, \mathbf{h}; \boldsymbol{\theta}) = \sum_i \frac{(v_i - b_i)^2}{2\sigma_i^2} - \sum_{ij} W_{ij} h_j \frac{v_i}{\sigma_i} - \sum_j a_j h_j \tag{131}$$

You should also use a smaller learning rate α compared to Bernoulli RBM.

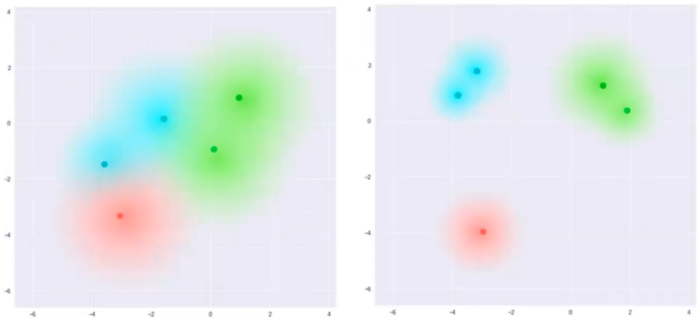
7 Variational Autoencoders

Variational autoencoders are very good at generating fake data/images. The general construction of VAEs, which bridges graphical models and deep learning, is based on generative probability models and does not really need to be implemented using neural nets. VAEs based on deep learning were proposed in 2013.

We start off by stating a fundamental problem with autoencoders. The latent space where the encoded vectors lie may not be contiguous or allow easy interpretation. For example, training an autoencoderon MNIST and then visualizing the encodings from a 2D latent space shows the formation of distinct clusters, but there are huge empty spaces (e.g. between 1 and 7) where the labeling may be ambiguous and not allow us to interpolate effectively.



Ideally, we want the encodings to be close to being contiguous while still being distinct. This allows smooth interpolation and enables construction of *new* samples. If the space has discontinuities and you sample a variation from there, the decoder will simply generate an unrealistic output.



Desired encoding Undesired encoding

The general idea here is to replace all the "point-estimates" into distributions in a regular autoencoder. In a way, we have done this already in softmax classification. In a classification neural network, it takes in an input $\mathbf{x}$ and outputs a softmax vector $NN(\mathbf{x}) = (p_1, \dots, p_K)^T$. This basically means that $NN(\mathbf{x}) = \boldsymbol{\theta}$ parameterizes the conditional distribution (in this case, multinomial) of $\mathbf{y}$ given $\mathbf{x}$.

$$Y \mid X = x \sim \text{Multinomial}(\boldsymbol{\theta} = NN(\mathbf{x}))$$

This is a much more efficient way to store conditional distributions than a $\dim(X)(K-1)$ lookup table.

7.1 Deep Latent Variable Models

Latent variable models that uses some deep learning architecture is pretty much what DLVMs are. This is essentially what we want to extend. We take a latent model $(\mathbf{x}, \mathbf{z})$ and want to approximate $p(\mathbf{x}, \mathbf{z})$. There are essentially 2 things we're interested in:

1. **Generation**: Computing $p(\mathbf{x} \mid \mathbf{z})$.
2. **Inference**: Computing $p(\mathbf{z} \mid \mathbf{x})$.

Given a latent sample $\mathbf{z}$ we want to find the conditional probability distribution of $\mathbf{x}$ given $\mathbf{z}$. We can also assume a simple prior $p(\mathbf{z})$ and calculate

$$p(\mathbf{x}, \mathbf{z}) = p(\mathbf{x} \mid \mathbf{z})\, p(\mathbf{z})$$

That is, we generate a latent variable from $p(\mathbf{z})$ and want to use this value to get the parameters of $x \sim p_{\boldsymbol{\theta}}(\mathbf{x} \mid \mathbf{z})$ with some decoder neural network $D_{\boldsymbol{\theta}}$ that parameterizes the distribution. Clearly put, $D_{\boldsymbol{\theta}}(\mathbf{z})$ are the parameters of $\mathbf{X} \mid \mathbf{Z} = \mathbf{z}$.

Example 7.1 (Bernoulli Random Vector)

We would like to approximate a D-dimensional Bernoulli vector $\mathbf{x}$ with a latent variable $\mathbf{z} \in \mathbb{R}^K$. We will assume a prior $p(\mathbf{z}) \sim N(\mathbf{0}, \mathbf{I})$, and let us have a neural net $\mathcal{D}_{\boldsymbol{\theta}}$ that parameterizes the random vector $\mathbf{x}$, where $x_i \sim \text{Bernoulli}(p_i)$ for p_i. Then,

$$p(\mathbf{x} \mid \mathbf{z}) = \prod_{d=1}^{D} p(x_d \mid \mathbf{z}) = \prod_{d=1}^{D} p_d^{x_d}(1-p_d)^{1-x_d} = \prod_{d=1}^{D} [\mathcal{D}_{\boldsymbol{\theta}}(\mathbf{z})]_d^{x_d} \left(1 - [\mathcal{D}_{\boldsymbol{\theta}}(\mathbf{z})]_d\right)^{1-x_d}$$

and we can see that since $\mathbf{p}$ has the flexibility of whatever vector in $[0,1]^D$ it can be captured by the neural net $\mathcal{D}$. It encompasses a broad family of Bernoulli probability distributions.

From the example above, we can see that we have some method to compute $p(\mathbf{x} \mid \mathbf{z})$. We train a neural net (somehow) and do forward prop on it to generate the correct parameters modeling the distribution of $\mathbf{x}$. However, computing

$$p(\mathbf{x}) = \int p(\mathbf{x}, \mathbf{z})\, d\mathbf{z} = \int p_{\boldsymbol{\theta}}(\mathbf{x} \mid \mathbf{z})\, p(\mathbf{z})\, d\mathbf{z}$$

is computationally intractable (note that in RBMs the conditional independence allowed us to integrate over $\mathbf{z}$ easily). To see why, in the example above, the integral becomes

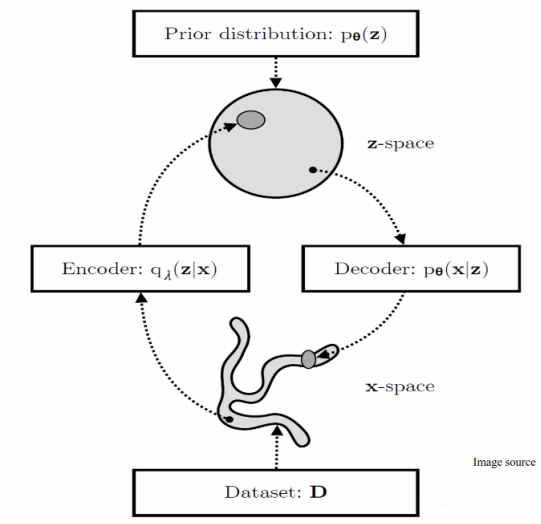
$$p(\mathbf{x}) = \sum_{\mathbf{z} \in \{0,1\}^K} \underbrace{\prod_{d=1}^{D} [\mathcal{D}_{\boldsymbol{\theta}}(\mathbf{z})]_d^{x_d} \left(1 - \mathcal{D}_{\boldsymbol{\theta}}(\mathbf{z})]_d\right)^{1-x_d}}_{\text{complex}} p(\mathbf{z})\, d\mathbf{z}$$

and integrating over all $\mathbf{z}$'s for more complex spaces is not feasible. Now let's focus on $p(\mathbf{z}|\mathbf{x})$. By Bayes rule, we can calculate

$$p(\mathbf{z} \mid \mathbf{x}) = \frac{p(\mathbf{x} \mid \mathbf{z}) p(\mathbf{z})}{p(\mathbf{x})}$$

Where we have established the intractability of the denominator. The first thing that comes to mind is to just do MCMC since $p(\mathbf{z} \mid \mathbf{x}) \propto p(\mathbf{x} \mid \mathbf{z}) p(\mathbf{z})$, but the forward propagation is too slow to sample efficiently. So, we must use our other trick in the book: **variational Bayes/inference**.

To do this, we construct another family of probability distributions parameterized by λ: $\{q_\lambda(\mathbf{z} \mid \mathbf{x})\}$, and we want to find a λ such that $q_\lambda(\mathbf{z} \mid \mathbf{x}) \approx p_{\boldsymbol{\theta}}(\mathbf{z} \mid \mathbf{x})$. Just like the generation model, we can build another neural network $\mathcal{E}_\phi$ such that $\lambda = \mathcal{E}_\phi(\mathbf{x})$ parameterizes the conditional distribution of $\mathbf{z}$. Essentially we are trying to construct an encoder and a decoder, which can be represented by the diagram below.



If $q_\lambda = p_{\boldsymbol{\theta}}$, then the diagram commutes, i.e. $p(\mathbf{z})p_{\boldsymbol{\theta}}(\mathbf{x} \mid \mathbf{z}) = p(\mathbf{x})p_{\boldsymbol{\theta}}(\mathbf{z} \mid \mathbf{x}) = p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})$.

Example 7.2 ()

If $\lambda = (\boldsymbol{\mu}, \boldsymbol{\sigma})$, where $\boldsymbol{\sigma}$ is just the vector representing variances of independent Gaussians, then we can use the neural network $\mathcal{E}$ to get

$$\lambda = \text{EncoderNN}_\phi(\mathbf{x}) = \mathcal{E}_\phi(\mathbf{x})$$

In the example, $\lambda = (\boldsymbol{\mu}, \log \boldsymbol{\sigma}^2)$ since we want to allow negative values, and $q_\lambda(\mathbf{z} \mid \mathbf{x}) = N(\mathbf{z} \mid \mathcal{E}_\phi(\mathbf{x})) = N(\mathbf{z};\, \boldsymbol{\mu}, \boldsymbol{\sigma}^2)$.

Now, just like in RBMs and really any density estimation problem, our job is to maximize the log likelihood of the training set:

$$\sum_t \log p(\mathbf{x}^{(t)})$$

In order to do this for this problem, we need a little fact to help us:

Theorem 7.1 ()

We have

$$KL\big(q_\lambda(\mathbf{z} \mid \mathbf{x}) \,||\, p_{\boldsymbol{\theta}}(\mathbf{z} \mid \mathbf{x})\big) = \mathbb{E}_{q_\lambda(\mathbf{z}|\mathbf{x})}[\log q_\lambda(\mathbf{z} \mid \mathbf{x})] + \log p_{\boldsymbol{\theta}}(\mathbf{x}) - \mathbb{E}_{q_\lambda(\mathbf{z}|\mathbf{x})}[\log p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})]$$

and hence

$$\log p(\mathbf{x}) = KL\big(q_\lambda(\mathbf{z} \mid \mathbf{x}) \,||\, p_{\boldsymbol{\theta}}(\mathbf{z} \mid \mathbf{x})\big) + \mathbb{E}_{q_\lambda(\mathbf{z}|\mathbf{x})}[\log p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})] - \mathbb{E}_{q_\lambda(\mathbf{z}|\mathbf{x})}[\log q_\lambda(\mathbf{z} \mid \mathbf{x})]$$

Proof.

TBD

So I have to maximize $\log p(\mathbf{x})$ with what we have derived just now, but the KL divergence part is intractable, since $p_{\boldsymbol{\theta}}(\mathbf{z} \mid \mathbf{x})$ is intractable. That is the entire reason we chose q_λ! Using the fact that the KL divergence is always greater than or equal to 0, we can drop the term and set a lower bound on the log likelihoods. This

lower bound is called the **variational lower bound**.

$$\sum_{i=1}^{N} \log p_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}) \geq \sum_{i=1}^{N} \mathbb{E}_{q_\lambda(\mathbf{z}|\mathbf{x}^{(i)})}[\log p_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}, \mathbf{z})] - \sum_{i=1}^{N} \mathbb{E}_{q_\lambda(\mathbf{z}|\mathbf{x}^{(i)})}[\log q_\lambda(\mathbf{z} \mid \mathbf{x}^{(i)})] = \text{ELBO}$$

If we assume that no two data points share their latent variables with each other, then ELBO decomposes into the sum of

$$\begin{aligned}
\text{ELBO} &= \sum_{i=1}^{N} \text{ELBO}_i \\
&= \sum_{i=1}^{N} \mathbb{E}_{q_\lambda(\mathbf{z}|\mathbf{x}^{(i)})}[\log p_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}, \mathbf{z})] - \sum_{i=1}^{N} \mathbb{E}_{q_\lambda(\mathbf{z}|\mathbf{x}^{(i)})}[\log q_\lambda(\mathbf{z} \mid \mathbf{x}^{(i)})] \\
&= \underbrace{\mathbb{E}_{q_\lambda(\mathbf{z}|\mathbf{x}^{(i)})}[\log p_{\boldsymbol{\theta}}(\mathbf{x}^{(i)} \mid \mathbf{z})]}_{\substack{\text{likelihood term} \\ \text{(reconstruction part)}}} - \underbrace{KL(q_\lambda(\mathbf{z} \mid \mathbf{x}^{(i)}) \,||\, p(\mathbf{z}))}_{\substack{\text{closeness of encoding to } p(\mathbf{z}) \\ \text{(typically Gaussian)}}}
\end{aligned}$$

Typtically, $p(\mathbf{z})$ is chosen to be standard normal. This process is true regardless of it model classes $p_{\boldsymbol{\theta}}(\mathbf{x}^{(i)} \mid \mathbf{z})$ and $q_\lambda(\mathbf{z} \mid \mathbf{x})$ are given by deep neural nets or not. If it is a deep neural net, then it's called a deep latent model.

Now to compute gradients, let us denote the ELBO w.r.t. the decoder and encoder paramters as $\mathcal{L}_{\theta,\lambda}(\mathbf{x})$. Then, we can obtain the unbiased gradient w.r.t. θ as such:

$$\begin{aligned}
\nabla_\theta \mathcal{L}_{\theta,\lambda}(\mathbf{x}) &= \nabla_\theta \big\{ \mathbb{E}_{q_\lambda(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}, \mathbf{z})] - \mathbb{E}_{q_\lambda(\mathbf{z}|\mathbf{x})}[\log q_\lambda(\mathbf{z} \mid \mathbf{x})] \big\} \\
&= \mathbb{E}_{q_\lambda(\mathbf{z}|\mathbf{x})} \big[\nabla_\theta \{ \log p_[\theta(\mathbf{x}, \mathbf{z}) - \log q_\lambda(\mathbf{z} \mid \mathbf{x}) \} \big] \\
&\approx \nabla_\theta \big\{ \log p_\theta(\mathbf{x}, \mathbf{z}) - \log q_\theta(\mathbf{z} \mid \mathbf{x}) \big\} \\
&= \nabla_\theta \log p_\theta(\mathbf{x}, \mathbf{z})
\end{aligned}$$

where the step with the $\approx$ just indicates that we approximate the expectation with a sample of size 1 over some minibatch. However, taking the gradient w.r.t. λ is more complicated since we cannot put the gradient in the expectation (since we are deriving and integrating w.r.t. λ). Fortunately, for continuous RVs, the unbiased estimator of the gradient can be obtained through the **reparamaterization trick**, which is some change of variable.

7.1.1 Reparameterization Trick

7.2 Variational Autoencoders

In a VAE, the $q_\lambda(\mathbf{z} \mid \mathbf{x})$ is the encoder and the $p_{\boldsymbol{\theta}}(\mathbf{x} \mid \mathbf{z})$ is the decoder.

1. **Encoding Neural Network**: Upon observing $\mathbf{x}$, the neural network $\mathcal{E}$ outputs parameters λ.
2. **Decoding Neural Network**: Upon observing $\mathbf{z}$, the neural network $\mathcal{D}$ outputs parameters $\boldsymbol{\theta}$.

We want to optimize $(\boldsymbol{\theta}, \lambda)$. To generate new samples, we just sample from $p(\mathbf{z})$ (usually standard Gaussian) and use the decoder to sample from $\mathbf{x}$.

This can be extended to deep layers.

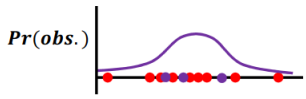
7.3 Conditional VAEs

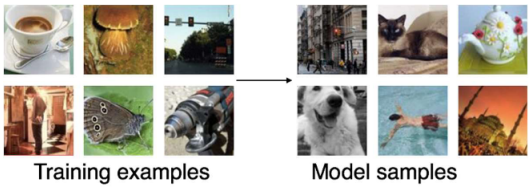
7.4 Importance Weighted Autoencoders

8 Generative Adversarial Networks

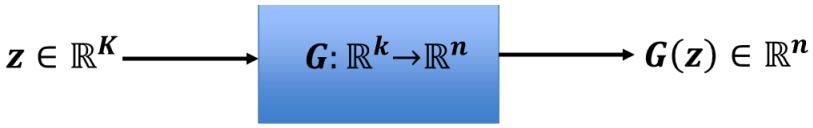
To introduce this topic, let's go back to the fundamentals talk about what the goal of modeling in general is. There exists in the real world a true distribution of the data we're interested in, and a generative model attempts to copy this distribution so that we can generate new synthetic observations. That is, density estimation really boils down to making sure that

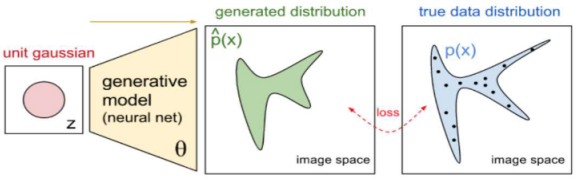
$$\mathbb{P}_{\text{observations}} \approx \mathbb{P}_{\text{syntheticobs.}}$$





The simplest types of densities can be analytically written and can be sampled from directly (e.g. Gaussian with inverse CDF or Box-Muller transform). More complicated models such as the RBM and VAE cannot be analytically written, but can still be directly sampled from with probabilities. GANs are implicit in the way that you can't estimate the probabilities but you can still sample from them. You essentially want to take some simple latent distribution and construct a differentiable a **Generator model** that maps it to a more complicated distribution in the observable space.

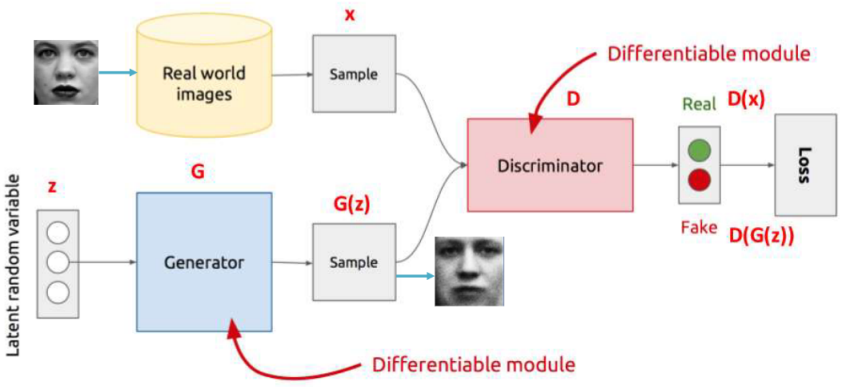




The problem with VAEs is that they tend to generate blurry images, which is the result of optimizing a variational lower bound rather than the true objective. This makes it easy to identify whether a given image is from the true dataset or has been artifically generated. In contrast, GANs generate high-resolution images by optimizing a pair of generator and disciminator neural networks, which play a game where one tries to bea the other.

1. The **generator** tries to generate fake samples to fool the discriminator. We sample from a latent space $\mathbf{z}$ and run that through the neural network to get $\mathbf{x} = \mathcal{G}_{\theta_g}(\mathbf{z})$.i It should be differentiable, but does not have to be invertible.
2. The **discriminator** tries to distinguish between real and fake samples, like a critic which can tell from real from fake. It should also be differentiable, and its output is essentially $0 \leq \mathcal{D}_{\theta_d}(\mathbf{x}) \leq 1$, with a value of 1 if real, 0 if fake.

We want to train these two models against each other, and in the end, we throw $\mathcal{D}$ away, since it's only role is to force $\mathcal{G}_{\theta_g}$ to work harder, which leaves us with a really good generative $\mathcal{D}_{\theta_g}$.



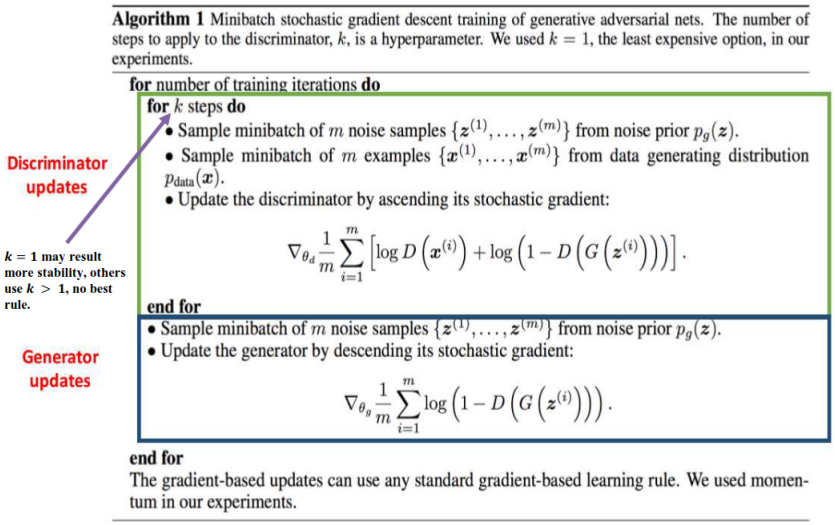
Now each of these networks will have its own set of parameters which we have to optimize. We want to optimize them by maximizing the likelihood such that the model says “real" to the samples from the world and “fake" to the generated samples. This leads to

$$\mathcal{L}(\theta_d, \theta_g) = V(\mathcal{D}, \mathcal{G}) = \underbrace{\mathbb{E}_{x\sim real}\left[\log \mathcal{D}_{\theta_d}(\mathbf{x})\right]}_{\substack{\text{log-prob that } \mathcal{D} \text{ correctly} \\ \text{predicts real data as real}}} + \underbrace{\mathbb{E}_{\mathbf{z}}\left[\log\left(1 - \mathcal{D}_{\theta_d}(\mathcal{G}_{\theta_g}(\mathbf{z}))\right)\right]}_{\substack{log-probthat\mathcal{D} \text{ correctly} \\ \text{predicts generated data as fake}}}$$

Therefore, the discrimiator is trying to maximize its reward (to get max value of 0), and the generator is trying to minimize the disciminator's reward (pulling this log probability down to $-\infty$). This is known as a **minimax** optimization problem, and we have to find

$$\min_{\theta_g} \max_{\theta_d} V(\mathcal{D}_{\theta_d}, \mathcal{G}_{\theta_g})$$

which is some saddle point. Note that the generator has no effect on the probability of $\mathcal{D}$ correctly identifying real images, so it only focuses on the latter term. The idea is to train both models simultaneously via SGD using mini-batches consisting of some generated samples and some real-world samples, which is called the **alternative gradient descent algorithm**.



It is usually the case that the generator does better than the disciminator, so we sometimes make $k > 1$ to allow more steps for training.

There is a vanishing gradient problem in GANs. For instance, assume that the model $\mathcal{G}_{\theta_g}$ has very bad parameters, and it generates very bad samples that the disciminator can detect very well. Then, $\mathbb{E}_{\mathbf{z}}\left[\log\left(1 -\right.\right.$

$\mathcal{D}_{\theta_d}(\mathcal{G}_{\theta_g}(\mathbf{z}))\big)\big]$ will be very close to 0, and the generator's cost will be very flat. Therefore, the gradient would die out and the generator can't improve! For example, if $\mathcal{D}$ was just a sigmoid function, then we can approximate the gradient of the expectation with a sample of the gradient, which would die out.

$$\begin{aligned}\nabla_{\theta_g} V(\mathcal{D},\mathcal{G}) &= \nabla_{\theta_g}\mathbb{E}_{z\sim q((\mathbf{z})}\big[\log\big(1-\mathcal{D}(\mathcal{G}(\mathbf{z}))\big)\big] \\ &\approx \nabla_a \log\big(1-\sigma(a)\big) \\ &= -\frac{\sigma(a)\big(1-\sigma(a)\big)}{1-\sigma(a)} = -\sigma(a) = -\mathcal{D}(\mathcal{G}(\mathbf{z}))\end{aligned}$$

and so the gradient goes to 0 if $\mathcal{D}$ is confident, i.e. $\mathcal{D}(\mathcal{G}(\mathbf{z}))\to 0$. Therefore, we can modify the cost for the generator term by changing the cost to

$$\mathbb{E}_{\mathbf{z}}\log\big(1-\mathcal{D}_{\theta_d}(\mathcal{G}_{\theta_g}(\mathbf{z}))\big)$$

and trying to minimize it.

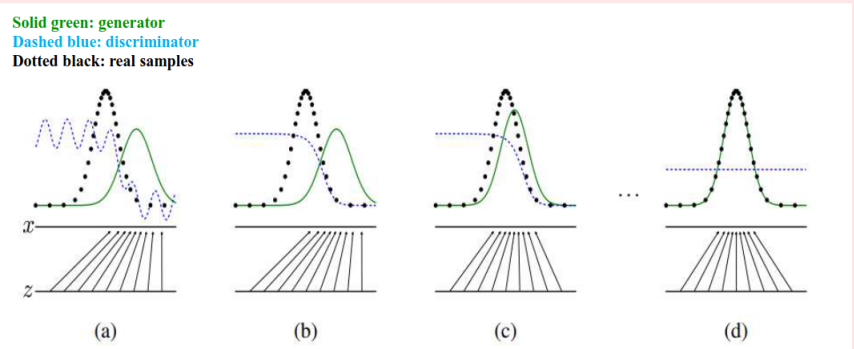
Theorem 8.1 (Nash Equilibrium)

Given the minimax loss above, for a fixed $\mathcal{G}$, the optimal discriminator $\mathcal{D}^*_G$ is given by

$$\mathcal{D}^*_G(\mathbf{x}) = \frac{p(\mathbf{x}\mid \text{real})}{p(\mathbf{x}\mid \text{real}) + p(\mathbf{x}\mid \text{synthesized})}$$

Therefore, the global minimum of the training criterion, $\max_{\mathcal{D}} V(\mathcal{D},\mathcal{G})$ is achieved if and only if

$$p(\mathbf{x}\mid \text{real}) = p(\mathbf{x}\mid \text{synthesized})$$



a) Near convergence. $Pr(x|synthesized)$ is similar to $Pr(x|real)$ and D is a partially accurate classifier.
b) D is trained to discriminate samples from data
c) After update to G, gradient of D has guided $G(z)$ to flow to regions that are more likely to be classified as data
d) Reaching an equilibrium point at which both distribution cannot improve because as (with enough capacity of D and G):

$$\boldsymbol{Pr(x|synthesized) = Pr(x|real)}$$

Proof.

We first have

$$V(\mathcal{D},\mathcal{G}) = \mathbb{E}_{\mathbf{x}\sim real}\log\mathcal{D}_{\theta_d}(\mathbf{x}) + \mathbb{E}_{\mathbf{z}}\log\big(1-\mathcal{D}_{\theta_d}(\mathcal{G}_{\theta_g}(\ddagger))\big)$$

Since $\mathcal{G}$ is fixed (i.e. θ_g is fixed) and acting, we can write the second expectation with respect to the

probability measure induced by $\mathcal{G}$.

$$\begin{aligned} V(\mathcal{D}, \mathcal{G}) &= \mathbb{E}_{\mathbf{x} \sim real} \log \mathcal{D}_{\theta_d}(\mathbf{x}) + \mathbb{E}_{\mathbf{w} \sim \text{fake}} \log \left(1 - \mathcal{D}_{\theta_d}(\mathbf{w})\right) \\ &= \int p(\mathbf{x} \mid \text{real}) \log \mathcal{D}_{\theta_d}(\mathbf{x})\, d\mathbf{x} + \int p(\mathbf{x} \mid \text{fake})\, \log \left(1 - \mathcal{D}_{\theta_d}(\mathbf{x})\right) d\mathbf{x} \end{aligned}$$

where the $\mathbf{x}$ in the second integral is a dummy variable. Taking the derivative w.r.t. $\mathbf{x}$ and setting equal to 0 gives

$$p(\mathbf{x} \mid \text{real}) \frac{1}{\mathcal{D}_{\theta_d}(\mathbf{x})} + p(\mathbf{x} \mid \text{fake}) \cdot \frac{-1}{1 - \mathcal{D}_{\theta_d}(\mathbf{x})} = 0$$

implies that

$$\mathcal{D}_{\theta_d}(\mathbf{x}) = \frac{p(\mathbf{x} \mid \text{real})}{p(\mathbf{x} \mid \text{real}) + p(\mathbf{x} \mid \text{fake})}$$

If the discriminator $\mathcal{D}$ is optimal, then the generator is minimizing the **Jensen-Shannon divergence** between the real and generated (model distributions). However, $\mathcal{D}$ is not optimal in practice since we have limited computational resources, the loss is non-convex, etc.

9 Normalizing Flows

We have seen many examples of generative models that attempt to produce a probability distribution p that approximates that of the data samples. Some are given by an explicit model (e.g. RBMs) while in others the model is implicit. The key idea for flow-based models[9] is that we want to map simple distributions (e.g. a Gaussian) to complex densities representing the data through an *invertible transformation*. Recall the lemma below from multivariate calculus.

Lemma 9.1 (Jacobi)

Let X, Z be absolutely continuous random variables in $\mathbb{R}^n$. Given that $f : \mathbb{R}^n \to \mathbb{R}^n$ is invertible and differentiable everywhere, with $X = f(Z), Z = f^{-1}(X)$, we claim

$$p_X(x) = p_Z(f^{-1}(x)) \cdot \left|(Df^{-1})(x)\right| \tag{132}$$

where $|(Df^{-1})(x)|$ is the determinant of total derivative of f^{-1} at x.

Proof.

For $n = 1$, we have

$$\begin{aligned} p_X(x) &= \frac{d}{dx} F_X(x) && (133)\\ &= \frac{d}{dx} F_Z(f^{-1}(x)) && (134)\\ &= p_z(f^{-1}(x)) \cdot \frac{d}{dx} f^{-1}(x) && (135)\\ &= p_z && (136) \end{aligned}$$

Therefore, if we parameterize f with some θ, then the marginal likelihood of x given θ can be written as

$$p_X(x;\ \theta) = p_Z(f_\theta^{-1}(x)) \cdot |(Df_\theta^{-1})(x)| \tag{137}$$

Therefore, if X is a complex distribution and Z is a simple one (e.g. uniform), we might have hope to efficiently compute $p_X(x)$ for any $x \in \mathbb{R}^n$ if

1. we can efficiently compute $f_\theta^{-1}(x)$, which allows us to compute $p_Z(f^{-1}(x))$ efficiently since Z is simple.
2. we can efficiently compute the Jacobian $Df^{-1}(x)$, and furthermore be able to compute the determinant efficiently.[10]

It looks like we have split this enormously hard problem of modeling X with three slightly less difficult problems. However, with some tricks, we may be able to get a simple enough parameterization of f to be able to compute its inverse plus the determinant of the Jacobian. As a first step, what if we add more intermediate functions? Consider the sequence of invertible functions $Z \xrightarrow{f} Y \xrightarrow{g} \to X$, where Z is a simple distribution that gets transformed to a slightly more complicated distribution Y that then gets transformed to a complex distribution X. We can apply the Jacobi theorem above to see $p_Y(y) = p_Z(f^{-1}(y)) \cdot |(Df^{-1})(x)|$, and so we have

$$\begin{aligned} p_X(x) &= p_Y(g^{-1}(x)) \cdot |(Dg^{-1})(x)| && (138)\\ &= p_Z(f^{-1}(g^{-1}(x))) \cdot |(Df^{-1})(y)| \cdot |(Dg^{-1})(x)| && (139)\\ &= p_Z((g \circ f)^{-1}(x))) \cdot |(Df^{-1})(y)| \cdot |(Dg^{-1})(x)| && (140) \end{aligned}$$

This leads to the following corollary.

[9] This has nothing to do with flow graphs and the max-flow-min-cut theorem in graph theory.
[10] Note that computing determinants are approximately $O(n^{2.4})$, which may not be practical.

Corollary 9.1 (Jacobi)

Given a sequence of transformations

$$Z = Z_0 \xrightarrow{f^1} Z_1 \xrightarrow{f^2} \dots \xrightarrow{f^{M-1}} Z_{M-1} \xrightarrow{f^M} Z_M = X \tag{141}$$

where each f^m is invertible and differentiable everywhere, let us denote $f = f^M \circ \cdot \circ f^1 : Z \to X$, which is invertible. Then

$$p_X(x) = p_Z(f^{-1}(x)) \cdot \prod_{m=1}^{M} \left|([Df^m]^{-1})(z_m)\right| \tag{142}$$

Therefore, by taking a sequence these functions f_θ^m, which may each have a simple parameterization, we may construct a very complex composition f_θ that may result in a very expressive Z. As for how we parameterize this, our notation will assume that $\theta = (\theta_1, \dots, \theta_M)$ and each θ_m parameterizes f^m. This is very similar to how a composition of linear mappings plus an activation gives us a very expressive neural network, and unsurprisingly, there is an analogue of the universal approximation theorem for transformations of this form.

Theorem 9.1 (Probability Integral Transform)

Any n-dimensional random variable in $\mathbb{R}^n$ that is absolutely continuous w.r.t. the Lebesgue measure can be constructed from the uniform distribution U on $[0,1]^n$. Since we can map to and back from invertibility, any two such random variables X and Y can be mapped from each other,

$$X \mapsto U \mapsto Y \tag{143}$$

The composition f_θ is pretty much our neural network, and we maximize the log-likelihood over the samples.

$$\theta^* = \underset{\theta}{\text{argmax}} \sum_{x \in \mathcal{D}} \log p_X(x) \tag{144}$$

$$= \underset{\theta}{\text{argmax}} \sum_{x \in \mathcal{D}} \log p_Z(f^{-1}(x)) \cdot \prod_{m=1}^{M} \left|([Df^m]^{-1})(z_m)\right| \tag{145}$$

10 Energy Based Models

11 Score Based Models

12 Graphical Models

13 Attention Models

We have seen the power of encoder decoder models, which can be used with a combination of RNNs or CNNs. Since we can plug and play different architectures, we can process and output different types of data.[11] CNNs were quite strong and have really no problem with scaling ever since the ResNet architecture, but historically, RNNs had two main problems.

1. They compute sequentially, since the hidden states must be a function of previous hidden states. This results in a linear time complexity, which is not ideal.
2. By encoding the entire sequence in a hidden latent space of dimension h, we are essentially trying to compress a possibly very long sequence into a single vector of predetermined dimension. This causes a *bottleneck problem* and words that are further away may be "forgotten."

We will see that attention solves the second bottleneck problem, and self-attention solves the sequential problem.

In general, feed forward networks treat features as independent, convolutional networks focus on relative location and proximity, and RNNs have tend to read in one direction. This may not be the most flexible way to process data, and we have some other problems.

1. When processing images, we may want our CNN to focus on a specific part of the image. For example, when we see a cat in the corner, other parts of the image does not matter, and we can have our CNN focus on the specific portion of the image containing the cat.
2. When reading sentences, different words may be interdependent, even if they are not next to each other, and so we may want to focus on different portions of a sentence (e.g. words 1 3, plus 10 15 which describes an object).

This is where attention comes in to the rescue. Attention is, to some extent, motivated by how we pay visual attention to different regions of an image or correlate words in one sentence. Human attention allows us to focus on certain regions or portions of our data with "high resolution" while perceiving the surrounding data in "low resolution." In a nutshell, attention in deep learning can be broadly interpreted as a vector of importance weights. First, we will introduce attention in the general setting, then move onto its specific implementation in RNNs through the seq2seq attention model, and finally its application in CNNs through Vision transformers.

13.1 Seq2Seq with Attention

The idea of **attention** provides a solution to this bottleneck problem. Basically, we want to establish connections from the decoder to not just the last hidden state of the encoder, but to all of its nodes. Each encoder node represents some information about each word, and by taking some weighted sum of these nodes, we can choose which one to put this attention on.

Definition 13.1 (Score Function)

Before we begin, let's define a metric, called a **score function**, to determine how similar two words (more specifically, their embeddings) are. The two simplest ways to do this are:

1. the standard dot product.
$$\text{score}(\mathbf{x}, \mathbf{y}) = \mathbf{x} \cdot \mathbf{y} \tag{146}$$
2. cosine similarity.
$$\text{score}(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x} \cdot \mathbf{y}}{||\mathbf{x}||\,||\mathbf{y}||} \tag{147}$$

Now that we have the score defined, the heart of attention comes with the query, key, value model. The general idea is this: for every prediction we want to make (whether it is classifying something or generating

[11] CNNs can be used to encode or output images and RNNs can be used to encode or output sequential data.

a new output word/token), we want to get its respective query vector and use it to look through key-value dictionary in order to get the most relevant information from it. With this information, combined with the query and whatever other information we have, we can make a prediction.

Let's go through this step by step. What we want to do is associate every input hidden node $\mathbf{h}_t$ with a 2-tuple consisting of a key-value pair.

$$\mathbf{h}_t \mapsto (\mathbf{k}_t, \mathbf{v}_t) \tag{148}$$

and associate every hidden output node $\mathbf{s}_t$ with a query value.

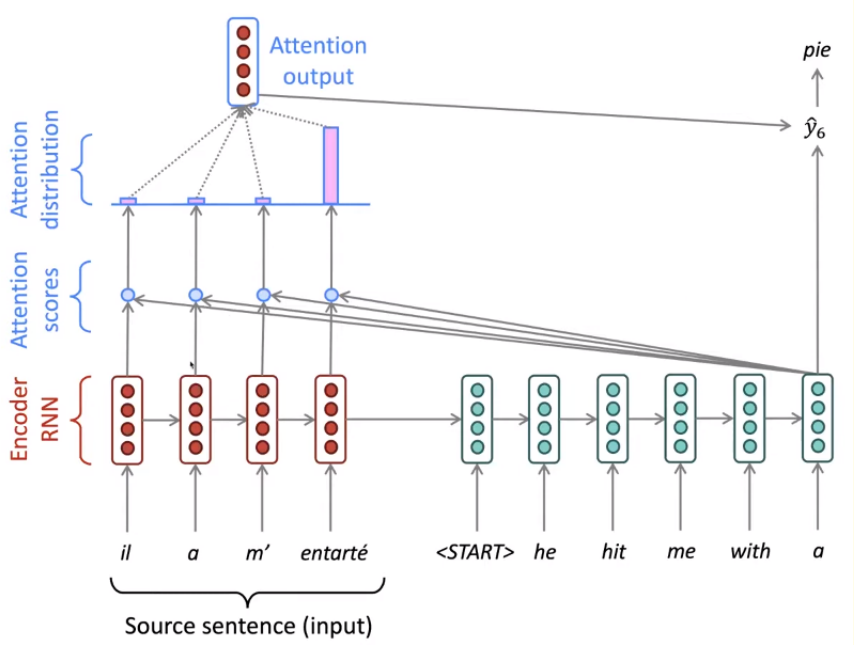
$$\mathbf{s}_T \mapsto \mathbf{q}_t \tag{149}$$

Definition 13.2 (Seq2Seq with Vanilla Attention)

Eventually, we would like to learn these key, value, queries, but for now let's focus on the forward propagation.

1. The input has been sequentially encoded and we have a special start token $\mathbf{s}_0$.
2. For $t' = 0$ until the $\mathbf{s}_{T'}$ is the end token, do the following.
   (a) Take the query $\mathbf{s}_i$ and compute the attention score $\text{score}(\mathbf{s}_i, \mathbf{h}_t)$ for all $t = 1, \dots, T$. This determines which encoder hidden state we should pay attention to.
   $$\mathbf{e}^{t'} = \big[\text{score}(\mathbf{s}_{t'}, \mathbf{h}_1), \dots, \text{score}(\mathbf{s}_{t'}, \mathbf{h}_T)\big] \in \mathbb{R}^T \tag{150}$$
   (b) We take its softmax to get the **attention distribution** $\boldsymbol{\alpha}^{t'}$ for this step (a discrete probability distribution)
   $$\boldsymbol{\alpha}^{t'} = \text{softmax}(\mathbf{e}^{t'}) \in \mathbb{R}^T \tag{151}$$
   (c) We use $\boldsymbol{\alpha}^{t'}$ to take a weighted sum of the encoder hidden states to get the attention output $\mathbf{a}_t$
   $$\mathbf{a}_{t'} = \sum_{t=1}^{T} \alpha_t^{t'} \mathbf{h}_t \in \mathbb{R}^h \tag{152}$$
   which acts as our context vector $\mathbf{C}_{t'}$ that we can now use in our vanilla seq2seq model.[a] Note that this context vector is different for every $\mathbf{s}_{t'}$, so at every step we can choose which encoder states/words to focus on. [b]

Another trick to improve performance is that these attention context vectors can be concatenated with the previous decoder hidden state to get more information in the already decoded part of the sentence.

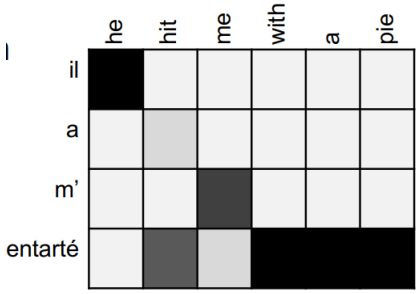


[a]This weighted sum is a selective summary of the information contained in the values, where the query determines which values to focus on. Attention is a way to obtain a fixed size representation of an arbitrary set of representations (the values), dependent on some other representation (the query).

[b]This is similar to a hash map where you have a set of key-value pairs. When you have a query, you want to search through the keys to see if it matches the query, and then it returns the value of the matched key. In our case, we have a query, and rather than looking for exact matches, we want to return a similarity score of the query q_i across all k_i's and provide a weighted sum of the corresponding values.

Overall, attention is extremely useful in improving all performance, and it is intuitive with how humans analyze things, too. It significantly improves neural machine translation by allowing the decoder to focus on certain parts of the source. It also provides more "human-like" model of the machine translation process (you can look back at the source sentence while translating, rather than needing to remember it all). It solves the bottleneck problem and helps with the vanishing gradient problem with these pseudo-residual connections through the context vector.

Finally, it provides some interpretability, as we can inspect the attention distribution to see what the decoder was focusing on (which again, we've never set explicitly but was learned by the model).



Code 13.1 ()

For an implementation of this in PyTorch, look here.

Now let's talk about how these parameters are already learned. The parameters of the model are.

1. The encoding matrices for the usual seq2seq model: $\mathbf{W}_e, \mathbf{U}_e$.
2. Generating key, value, and query vectors for every possible embedding is not practical.[12] A more compact way to represent them are through linear maps, so we are learning matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ such that

$$\begin{aligned} \mathbf{q}_{t'} &= \mathbf{Q}\mathbf{s}_{t'} \\ \mathbf{k}_t &= \mathbf{K}\mathbf{h}_t \\ \mathbf{v}_t &= \mathbf{V}\mathbf{h}_t \end{aligned}$$

3. We should now have the decoding matrices $\mathbf{W}_d$ that takes in the attention context vector (plus maybe the previous hidden decoder state) and the original matrix $\mathbf{U}_d$.

Therefore, this problem reduces to learning the matrices $\mathbf{W}_e, \mathbf{U}_e, \mathbf{Q}, \mathbf{K}, \mathbf{V}, \mathbf{W}_d, \mathbf{U}_d$.

Example 13.1 (Score Functions)

Having additional flexibility with the score functions can also improve learning. We provide more examples here, which give more parameters to learn as well.

1. The general attention model allows us to train a shared-weight matrix, allowing for $s \neq h$.

$$e_t^{t'} = \text{score}(\mathbf{s}_{t'}, \mathbf{h}_t) = \mathbf{s}_{t'}^T \mathbf{W}_a \mathbf{h}_t \in \mathbb{R}$$

   However, it may seem like there are too many parameters in $\mathbf{W}_a$, having to learn sh values.
2. The reduced rank multiplicative attention uses low rank matrices, allowing us to learn only $ks + kh$ parameters for matrices $\mathbf{U}_a \in \mathbb{R}^{k \times s}, \mathbf{V}_a \in \mathbb{R}^{k \times h}$ where $k << s, h$.

$$e_t^{t'} = \text{score}(\mathbf{s}_{t'}, \mathbf{h}_t) = \mathbf{s}_{t'}^T (\mathbf{U}_a^T \mathbf{V}_a) \mathbf{h}_t = (\mathbf{U}_a \mathbf{s}_{t'})^T (\mathbf{V}_a \mathbf{h}_t) \in \mathbb{R}$$

3. Additive attention uses a neural net layer defined

$$e_t^{t'} = \mathbf{v}_a^T \tanh(\mathbf{W}_a \mathbf{h}_t + \mathbf{V}_a \mathbf{s}_{t'}) \in \mathbb{R}$$

   where $\mathbf{W}_a \in \mathbb{R}^{r \times h}, \mathbf{V}_a \in \mathbb{R}^{r \times s}$ are weight matrices, $\mathbf{v}_a \in \mathbb{R}^r$ is a weight vector, and r (the attention dimensionality) is a hyperparameter.

This can be naturally extended to other architectures, as we will explore later.

Example 13.2 (Images)

Given an image of size 224×224, we can make patches of size 16×16, and then flatten them to get a 196×768 matrix with a 2-dimensional positional encoding scheme. We can then apply a linear transformation to get the query, key, and value vectors.

13.2 Self-Attention Layer

While we have solved the bottleneck problem, this entire process is still sequential since every hidden decoder node requires us to know the previous hidden node. There are two sequential processes in the regular seq2seq attention model.

1. The sequential encoding of the input sentence.
2. The sequential decoding of the output sentence. This unfortunately is not possible in transformers to parallelize.

[12] You would need three $d_{\text{embedding}} \times |\mathcal{V}|$ matrices, where $\mathcal{V}$ is the set of our vocabulary, which can go easily past 500,000 elements.

We will focus on parallelizing the first part by temporarily forgetting about encoder-decoder models and just thinking about how to incorporate a good encoder with attention that is parallelizable. This extension is quite simple. Let $\mathbf{w}_{1:n}$ be a sequence of words in vocabulary $\mathcal{V}$. The key here is that rather than inputs having key-values and outputs having queries, *all* words are associated with a 3-tuple of key, value, query.

$$\mathbf{x}_i \mapsto (\mathbf{q}_i, \mathbf{k}_i, \mathbf{v}_i) \tag{153}$$

Definition 13.3 (Standard Scaled Dot-Product Attention)

If we keep the score function to be the dot-product, the derivations become quite simple.

1. For each $\mathbf{w}_i$, let $\mathbf{x}_i = E\mathbf{w}_i$ be the word embedding (with $E \in \mathbb{R}^{d \times |\mathcal{V}|}$ the embedding matrix).
2. We transform each word embedding with the (learned) weight matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V}$.

$$\begin{aligned} \mathbf{q}_i = \mathbf{Q}\mathbf{x}_i &\implies Q = \mathbf{Q}X \\ \mathbf{k}_i = \mathbf{K}\mathbf{x}_i &\implies K = \mathbf{K}X \\ \mathbf{v}_i = \mathbf{V}\mathbf{x}_i &\implies V = \mathbf{V}X \end{aligned}$$

   where $X = [x_1, \dots, x_n] \in \mathbb{R}^{d \times n}$.
3. We compute pairwise similarities between keys and queries and normalize with the softmax to get the attention distribution for each word.

$$\mathbf{e}_{ij} = \mathbf{q}_i^T \mathbf{k}_j, \quad \boldsymbol{\alpha}_{ij} = \frac{\exp(\mathbf{e}_{ij})}{\sum_{j'} \exp(\mathbf{e}_{ij'})} \tag{154}$$

4. Compute the output for each word as a weighted sum of values.

$$\mathbf{o}_i = \sum_j \boldsymbol{\alpha}_{ij} \mathbf{v}_j \tag{155}$$

Ultimately, this can be parallelized into one matrix operation.[a]

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{E_k}}\right)V \tag{156}$$

where the softmax is done to each row.[b][c]

Therefore, when you do a forward pass on an attention layer with input x, you first get the query vector q, extract the attention-weighted values from the key-value dictionary, and then return the weighted sum of the values. To give explicit parameterizations using the query, key, value encoding matrices, we can write this as

$$\text{Attention}(x\,;\, \mathrm{Q}, \mathrm{K}, \mathrm{V}) = \text{softmax}\left(\frac{(\mathbf{Q}x)(\mathbf{K}x)^T}{\sqrt{E_k}}\right)(\mathbf{V}x) \tag{157}$$

This will give us a vector $\mathbf{o}_{1:n}$ consisting of the encoded vectors for each word in the sentence, and best of all, this is parallelizable!

[a] Note that in order to even do such a thing, we must know n beforehand. This can be solved by simply fixing some maximum length, padding everything to be some null token after the end token, and masking all the null tokens to be 0. More on masking later.

[b] We divide by $\sqrt{E_k}$ to stabilize the gradients since as dimensionality increases, the dot product between random vectors tend to get large, leading to large softmax inputs. You can simply compute the variance of two d-dimensional Gaussian vectors and see that their variances scales linearly with d.

[c] You can see that if we have simple dot-product similarity scores, then $E_k = E_v$, but this need not be true in general. We will explore other similiarity score in the next subsection.

There are three problems however.

1. This self-attention encoding does not account for the position/order of the words. Therefore, some

positional embedding is needed.

2. Our plan is to stack this layer multiple times on top of each other. However, we are just composing linear maps ultimately, so some nonlinearity is needed.
3. To use self-attention in *decoders*, as we will see later on, we don't want to have any attention on later parts of a sentence, so we need some way to *mask* future words.

We will deal with the first two problems and address the third problem in the transformer archictecture.

13.2.1 Tokenization and Positional Embeddings

Given an input (or an output) $\mathbf{x}$, it must be tokenized into a sequence of tokens. This is a general preprocessing step that is done for any input, whether it be a sequence of words, a sequence of regions in an image, or a sequence of anything. The raw token data will be denoted w_i for $i = 1, \dots, n$.We can then embed these tokens into a vector space $x_i \in \mathbb{R}^d$.

As we will see later, attention does not have a way to discern the order of the input sequences. Therefore, we must add this positional information to the encoding. The most obvious way would be to simply concatenate the position of the token to the end with an index.

$$x_i \mapsto [x_i, i] \tag{158}$$

However, this is not ideal since this tends to corrupt the embedding of the token. Instead, we can think of adding certain vectors representing components to the original embedding.

$$x_i \mapsto x_i + p_i \tag{159}$$

Certain ways come to mind, such as simply letting p_i be the vector of all i's. This tends not to work in progress since the values of i get too large and corrupts the embeddings too much.[13] Normalizing the values of i to be in $[0, 1]$ is disadvantageous because now the positional embedding p_i is dependent on the length of the total input sequence. Therefore, we need two properties:

1. The positional encoding should be independent of the input sequence length.
2. The positional encoding shouldn't be too large that it corrupts the semantic meaning behind the original embedding.

It turns out that the sinusoidal function satisfies these properties.

Definition 13.4 (Sinusoidal Position Embedding)

Given the embeddings $x_i \in \mathbb{R}^d$, we can add a positional encoding to it by

$$x_i \mapsto x_i + p_i$$

where the positional encoding is given by the vector where each component is defined as

$$(p_i)_j = \begin{cases} \sin\left(\frac{i}{10000^{2j/d}}\right) & \text{if } j \text{ is even} \\ \cos\left(\frac{i}{10000^{2j/d}}\right) & \text{if } j \text{ is odd} \end{cases} \tag{160}$$

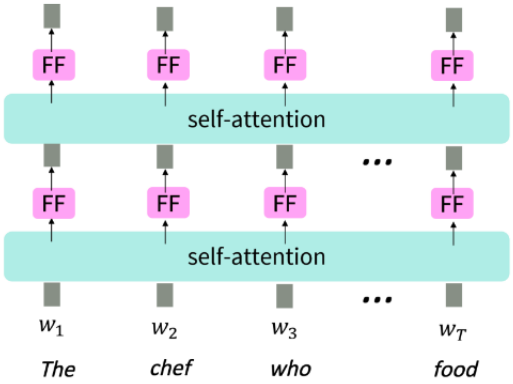
where i iterates through the tokens and j iterates through the dimensions of the embedding.

13.2.2 Stacked Attention Layers and Multi-Head Attention

The second problem of introducing nonlinearities is quite simple. Once we have the output of the first self-attention layer $\mathbf{o}_{1:n} = [\mathbf{o}_1, \dots, \mathbf{o}_n]$, we can just input each $\mathbf{o}_i$ through a small MLP to introduce nonlinearity before inputting it into the next self-attention layer.

[13] A helpful Medium article here

$$m_i = MLP(\text{output}_i) = W_2 * \text{ReLU}(W_1 \times \text{output}_i + b_1) + b_2$$



Boom, problem solved.

Going back, if we want to look at the attention for token x_i, we want to look through all $q_i^T k_j$ for all j and find out where it is high. But perhaps we want to focus on different j for different reasons. The following example may illustrate why.

Example 13.3 (Semantic and Syntactic Attention)

Given the sentence *I went to the bank and got some money.*, one type of attention may look at the semantic meaning of the words, such as associating *bank* with *money*. However, we may also want to look at the syntactic meaning of the words, such as associating *went* with *bank*. When we read sentences, we have different types of attention for different reasons, and so having multiple heads of attention may be useful.

Definition 13.5 (Multi-Head Attention)

Therefore, let us construct multiple attention heads by defining multiple triplets of (Q, K, V) matrices. This may be more computationally inefficient, so we simply scale down the size of these matrices from

$$Q \in \mathbb{R}^{E_q \times d}, K \in \mathbb{R}^{E_k \times d}, V \in \mathbb{R}^{E_v \times d}$$

to

$$Q_\ell \in \mathbb{R}^{E_q \times d/h}, K_\ell \in \mathbb{R}^{E_k \times d/h}, V_\ell \in \mathbb{R}^{E_v \times d/h}$$

where h is the number of heads. We are essentially decreasing the size of the token embedding dimension in order to get more heads. We can then do attention on each head separately.

$$\text{Attention}_\ell = \text{Attention}(Q_\ell, K_\ell, V_\ell) = \text{softmax}\left(\frac{Q_\ell K_\ell^T}{\sqrt{E_k/h}}\right) V_\ell \tag{161}$$

and we can simply concatenate them together to get the final output.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{Attention}_1, \dots, \text{Attention}_h) W^O$$

where $W^O \in \mathbb{R}^{d \times E_v}$ is a learnable weight matrix that mixes these heads together with a final linear transformation.[a] This entire process is shown in Figure 9.

[a] There is a valid concern that these heads may all end up learning the same thing and may just converge onto the same thing. However, this is not what happens in practice.

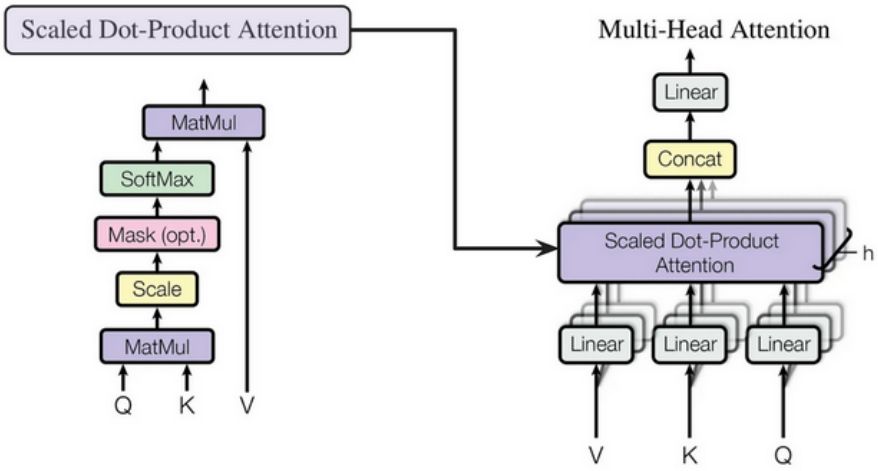


Figure 9: Diagram of multi head attention.

13.3 Transformers

With self-attention out of the way, the transformer architecture becomes quite simple. An overview of it is shown in Figure 10.

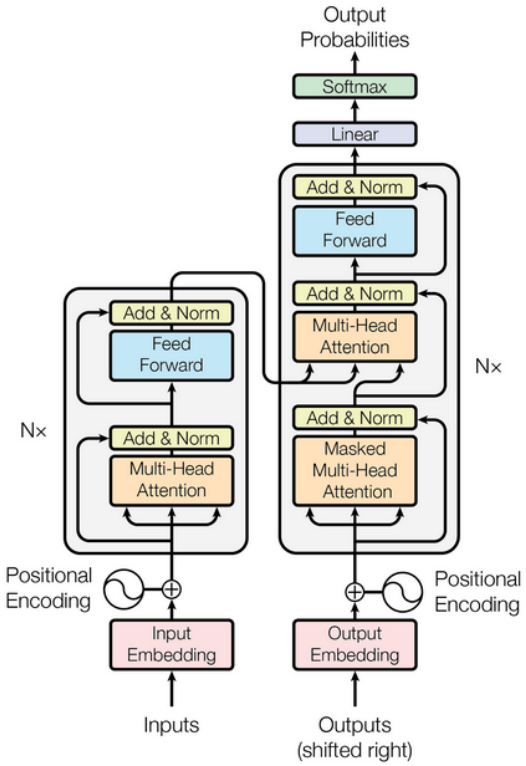


Figure 10: Transformer architecture.

The encoder is quite simple. You take the input embedding and add the positional embeddings to get $\{x_i \in \mathbb{R}^d\}_{i=1}^n$. You then pass it through a multi-head self-attention layer, which has outputs of shape $E_v \times n$, and then pass it through a feed forward network, adding residual connections and normalization

layers to help with training. You then repeat this process N times, which gives out the encoded sequence $\mathbf{h}_{1:n} = [\mathbf{h}_1, \dots, \mathbf{h}_n]$, now ready to be fed into the decoder.[14]

The decoder has two different self-attention layers. First, we run the generated output sequence through a masked self-attention layer, which generates the hidden nodes $\mathbf{z}_{1:n} = [\mathbf{z}_1, \dots, \mathbf{z}_n]$ representing the state of the currently decoded sentence. Again, we have some maximum output length to ensure that we are working with a fixed size, and manually mask all tokens after the current one to be 0.

Then, another **cross-attention** layer takes both $\mathbf{h}_{1:n}$ and $\mathbf{z}_{1:n}$ and with its trained $(\mathbf{K}, \mathbf{V}, \mathbf{Q})$, computes the key, value, and query matrices as

$$K = \mathbf{K}\mathbf{h}_{1:n}, \quad V = \mathbf{V}\mathbf{h}_{1:n}, \quad Q = \mathbf{Q}\mathbf{z}_{1:n}, \tag{162}$$

now ready to be plugged into to the self-attention formulas, integrating both the inputs and the current output to generate the result. This again outputs another list of vectors, which are run through an MLP and then have another set of $(\mathbf{K}, \mathbf{V}, \mathbf{Q})$ matrices waiting for them. This makes sense, since we want to use the output sequence to query the input key-values and attend to the correct set of tokens.

The output of this is then passed through a feed forward network, with some residual connections and normalization, and finally a linear layer transforms the output dimensions to whatever is needed (e.g. size of the vocabulary, or number of classes). Once this is done, a new word is generated,[15] and this word (along with all previous words) is now used as the new input to the decoder in place of the start token. This process is done until the stop token is generated by the decoder. Notice that we encode with a bidirectional model (no masking) and generated the target with a unidirectional model (masking).

Note that again, parallelization of the decoder is not possible in the transformer architecture. Additionally, you can see that more normalization layers and residual connections are needed to train efficiently. This is very important in practice.

Despite all its advantages, self-attention has quadratic runtime complexity with respect to the sequence length since we need to compute attention for all pairs of words. This is worse than the linear runtime complexity in RNNs.

13.3.1 Masking

The final aspect we did not address is the masking. When we are training the transformer on a corpus of data, the decoder first computes self-attention on all the previous outputs first to get the query, and then takes in the output of the encoder self-attention layer as the keys and values. Then it does self-attention once more over these triplets, essentially doing a self-attention layer over the entire input and all tokens up to the current decoded output.

When training this model, we have access to the entire decoded output, and we want to make sure that we do not perform self-attention on any future words since it will most likely attend 100% to the next word to generate the next word! This does not learn anything, so we artificially set the attention distribution for all future output words to be 0. This is usually done by setting the attention scores to $-\infty$ (or more practically, a very negative number) which will result in 0 after softmaxing.[16]

13.4 Practical Implementation

In here we go over the nitty gritty details that comes into implementing a transformer in `pytorch==2.3.0`.

13.4.1 Key, Value, Query Matrices and In Projections

The first thing is that these key, value, query does not have to necessarily equal to the dimension embedding, which we will denote as E. One flexibility is that we don't necessarily need to set the dimensions of the keys,

[14] You can see that to support iterating through n times, E_v should equal d.

[15] Note that we have not specified how to get the corresponding word given an embedding vector. This is not within the scope of these notes and are covered in my natural language processing notes.

[16] Here is a nice explanation here.

values, and queries the same. We can see in the constructor of the `torch.nn.MultiheadAttention` module that you can input your own dimensions for the keys and values, but queries must be the same as E.

```
def __init__(self, embed_dim, num_heads, dropout=0., bias=True, add_bias_kv=False,
    add_zero_attn=False,
            kdim=None, vdim=None, batch_first=False, device=None, dtype=None) -> None:
   ...
  self.embed_dim = embed_dim
  self.kdim = kdim if kdim is not None else embed_dim
  self.vdim = vdim if vdim is not None else embed_dim
  self._qkv_same_embed_dim = self.kdim == embed_dim and self.vdim == embed_dim
```

In fact, $K \in \mathbb{R}^{d_k \times E}$, $V \in \mathbb{R}^{d_v \times E}$, then $QK^T \in \mathbb{R}^{E \times d_k}$. Since this obviously leads to dimension mismatch problem when we multiply it with the matrix V, what we do is have an **in projection** layer that maps everything to dimension E. We can check this for the following.

$$K_{proj} \in \mathbb{R}^{E \times d_k}, V_{proj} \in \mathbb{R}^{E \times d_v}, Q_{proj} \in \mathbb{R}^{E \times E} \tag{163}$$

There are two ways to store these projection matrices, as shown in the constructor.

```
# in the constructor
...
self._qkv_same_embed_dim = self.kdim == embed_dim and self.vdim == embed_dim

if not self._qkv_same_embed_dim:
  self.q_proj_weight = Parameter(torch.empty((embed_dim, embed_dim), **factory_kwargs))
  self.k_proj_weight = Parameter(torch.empty((embed_dim, self.kdim), **factory_kwargs))
  self.v_proj_weight = Parameter(torch.empty((embed_dim, self.vdim), **factory_kwargs))
  self.register_parameter('in_proj_weight', None)
else:
  self.in_proj_weight = Parameter(torch.empty((3 * embed_dim, embed_dim), **factory_kwargs))
  self.register_parameter('q_proj_weight', None)
  self.register_parameter('k_proj_weight', None)
  self.register_parameter('v_proj_weight', None)
```

1. If these shapes are different, then we store them in separate matrices as above.

   ```
   att = nn.MultiheadAttention(embed_dim=50, num_heads=1, bias=False, kdim=30, vdim=40)
   att.q_proj_weight.shape # torch.Size([50, 50])
   att.k_proj_weight.shape # torch.Size([50, 30])
   att.v_proj_weight.shape # torch.Size([50, 40])
   ```

2. If these shapes are the same, then we just store them in a $3E \times E$ matrix by concatenation them.

   ```
   att = nn.MultiheadAttention(embed_dim=50, num_heads=1, bias=False)
   att.in_proj_weight.shape # torch.Size([150, 50])
   ```

These conditions are asserted throughout the forward pass as well.

13.4.2 Masking

We multiply by a masking matrix.

13.4.3 Computing Attention

First we reshape them so that they are batch first.

If `needs_weights = True`, we also output the attention weights in addition to the output, but it is said that this degrades performance. It is by default true but should be set to false for small tasks.

13.4.4 Forward Pass of MultiheadAttention

First, we should look at the main function that computes self-attention. We omit a large part of the code to focus on the relevant details.

```
# torch.nn.functional
def multi_head_attention_forward(
  query: Tensor,
  key: Tensor,
  value: Tensor,
  embed_dim_to_check: int,
  num_heads: int,
  in_proj_weight: Optional[Tensor],
  in_proj_bias: Optional[Tensor],
  bias_k: Optional[Tensor],
  bias_v: Optional[Tensor],
  add_zero_attn: bool,
  dropout_p: float,
  out_proj_weight: Tensor,
  out_proj_bias: Optional[Tensor],
  training: bool = True,
  ...
):
# first unsqueezes the input if it is not batched.

# look at the input dimensions and check that multiheads divide it evenly
#
assert embed_dim == embed_dim_to_check, \
    f"was expecting embedding dimension of {embed_dim_to_check}, but got {embed_dim}"
if isinstance(embed_dim, torch.Tensor):
    # embed_dim can be a tensor when JIT tracing
    head_dim = embed_dim.div(num_heads, rounding_mode='trunc')
else:
    head_dim = embed_dim // num_heads
assert head_dim * num_heads == embed_dim, f"embed_dim {embed_dim} not divisible by num_heads
    {num_heads}"
if use_separate_proj_weight:
    # allow MHA to have different embedding dimensions when separate projection weights are
    used
    assert key.shape[:2] == value.shape[:2], \
        f"key's sequence and batch dims {key.shape[:2]} do not match value's {value.shape[:2]}"
else:
    assert key.shape == value.shape, f"key shape {key.shape} does not match value shape
    {value.shape}"

# Computes in-projection, which is an affine map before doing attention.
# in_proj_weight = [W_q, W_k, W_v], in_proj_bias = [b_q, b_k, b_v]
# computes q = q * W_q + b_q, k = k * W_k + b_k, v = v * W_v + b_v
if not use_separate_proj_weight:
  q, k, v = _in_projection_packed(query, key, value, in_proj_weight, in_proj_bias)
else:
  if in_proj_bias is None:
      b_q = b_k = b_v = None
  else:
      b_q, b_k, b_v = in_proj_bias.chunk(3)
  q, k, v = _in_projection(query, key, value, q_proj_weight, k_proj_weight, v_proj_weight,
```

```
      b_q, b_k, b_v)

# prepare attention mask
# add bias along batch dimension
# more preparation with mask
...
# Now calculate attention
if need_weights:
  # scale q_scale for the sqrt(E) division factor
  B, Nt, E = q.shape
  q_scaled = q * math.sqrt(1.0 / float(E))

if attn_mask is not None:
  # torch.baddbmm is a pybinded C function implementing matrix multiplication
  # of form attn_mask + q_scaled @ k^T
  attn_output_weights = torch.baddbmm(attn_mask, q_scaled, k.transpose(-2, -1))
else:
  # torch.bmm is also a pybinded C function q_scaled + k^T
  attn_output_weights = torch.bmm(q_scaled, k.transpose(-2, -1))
...
# softmax it and then multiply it by V.
attn_output_weights = softmax(attn_output_weights, dim=-1)
attn_output = torch.bmm(attn_output_weights, v)

# final linear layer for more weightings.
attn_output = attn_output.transpose(0, 1).contiguous().view(tgt_len * bsz, embed_dim)
attn_output = linear(attn_output, out_proj_weight, out_proj_bias)
attn_output = attn_output.view(tgt_len, bsz, attn_output.size(1))

# optionally average attention weights over heads
attn_output_weights = attn_output_weights.view(bsz, num_heads, tgt_len, src_len)
if average_attn_weights:
    attn_output_weights = attn_output_weights.mean(dim=1)

if not is_batched:
    # squeeze the output if input was unbatched
    attn_output = attn_output.squeeze(1)
    attn_output_weights = attn_output_weights.squeeze(0)
return attn_output, attn_output_weights
```

This is precisely the function that is called in the forward method of the `MultiheadAttention` module.

13.4.5 Transformer

In the transformer, we can see that if we peek at the state dictionary, it composes of an encoder and a decoder, each with a certain number of attention layers. There are 6 attention layers each by default.

```
transformer = nn.Transformer()
transformer.state_dict
# output
<bound method Module.state_dict of Transformer(
  (encoder): TransformerEncoder(
    (layers): ModuleList(
      (0-5): 6 x TransformerEncoderLayer(
        (self_attn): MultiheadAttention(
          (out_proj): NonDynamicallyQuantizableLinear(in_feat
ures=512, out_features=512, bias=True)
        )
```

```
        (linear1): Linear(in_features=512, out_features=2048,
 bias=True)
        (dropout): Dropout(p=0.1, inplace=False)
        (linear2): Linear(in_features=2048, out_features=512,
 bias=True)
        (norm1): LayerNorm((512,), eps=1e-05, elementwise_aff
ine=True)
        (norm2): LayerNorm((512,), eps=1e-05, elementwise_aff
ine=True)
        (dropout1): Dropout(p=0.1, inplace=False)
        (dropout2): Dropout(p=0.1, inplace=False)
      )
    )
    (norm): LayerNorm((512,), eps=1e-05, elementwise_affine=T
rue)
  )
  (decoder): TransformerDecoder(
    (layers): ModuleList(
      (0-5): 6 x TransformerDecoderLayer(
        (self_attn): MultiheadAttention(
          (out_proj): NonDynamicallyQuantizableLinear(in_feat
ures=512, out_features=512, bias=True)
        )
        (multihead_attn): MultiheadAttention(
          (out_proj): NonDynamicallyQuantizableLinear(in_feat
ures=512, out_features=512, bias=True)
        )
        (linear1): Linear(in_features=512, out_features=2048,
 bias=True)
        (dropout): Dropout(p=0.1, inplace=False)
        (linear2): Linear(in_features=2048, out_features=512,
 bias=True)
        (norm1): LayerNorm((512,), eps=1e-05, elementwise_aff
ine=True)
        (norm2): LayerNorm((512,), eps=1e-05, elementwise_aff
ine=True)
        (norm3): LayerNorm((512,), eps=1e-05, elementwise_aff
ine=True)
        (dropout1): Dropout(p=0.1, inplace=False)
        (dropout2): Dropout(p=0.1, inplace=False)
        (dropout3): Dropout(p=0.1, inplace=False)
      )
    )
    (norm): LayerNorm((512,), eps=1e-05, elementwise_affine=T
rue)
  )
)>
```

13.5 Vision Transformers

We have hinted at attention being applicable in other architectures, and the most popular is in computer vision. Historically, CNNs were very useful because they take into account the locality and translational-invariance of objects in images inherently in the convolutions. This is a great strength of convolutional networks.

Can transformers beat this? These assumptions are not built into the architecture, and researchers were quite unsuccessful in passing the benchmarks set by CNNs, but it turned out that in 2020, with enough

training and a large enough architecture, *vision transformers* in fact did surpass CNNs.

14 Learning Methodologies

14.1 Student Teacher Models

14.2 Curriculum Learning

15 Adversarial Learning

16 Semi Supervised Learning

There has been good stream of work done at Google Brain that first came up with MixMatch in 2019 [2], which combined the state of the art semi supervised methods. This model was then improved the next year with ReMixMatch [1], and then improved again with a much more simple model called FixMatch [8].

16.1 Pseudo Label Learning

Generating psuedo labels on unlabeled datasets and training on them.

16.2 Consistency Regularization

Ensuring that the model is consistent with its predictions on certain inputs and neighbors of the inputs. It relies on the assumption that perturbed versions of the same input should have the same output. When wanting to make sure that the outputs are consistent with augmentations of the input, this is called **augmentation anchoring**. There are ways in which we use a combination of **weak augmentation** and **strong augmentation** to achieve this.

16.3 Distribution Alignment

16.4 Weak Supervision

References

[1] David Berthelot, Nicholas Carlini, Ekin D. Cubuk, Alex Kurakin, Kihyuk Sohn, Han Zhang, and Colin Raffel. Remixmatch: Semi-supervised learning with distribution alignment and augmentation anchoring, 2020.

[2] David Berthelot, Nicholas Carlini, Ian Goodfellow, Nicolas Papernot, Avital Oliver, and Colin Raffel. Mixmatch: A holistic approach to semi-supervised learning, 2019.

[3] An Mei Chen, Haw-minn Lu, and Robert Hecht-Nielsen. On the geometry of feedforward neural network error surfaces. *Neural Computation*, 5(6):910–927, 11 1993.

[4] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[5] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks, 2019.

[6] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[7] Diganta Misra. Mish: A self regularized non-monotonic activation function, 2020.

[8] Kihyuk Sohn, David Berthelot, Chun-Liang Li, Zizhao Zhang, Nicholas Carlini, Ekin D. Cubuk, Alex Kurakin, Han Zhang, and Colin Raffel. Fixmatch: Simplifying semi-supervised learning with consistency and confidence, 2020.

[9] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

[10] Edric Tam and David Dunson. Fiedler regularization: Learning neural networks with graph sparsity, 2020.