Multilayer Perceptrons

Muchang Bahng

July 25, 2023

Contents

There are two mainstream packages that impelement deep learning: Tensorflow and PyTorch. We will use PyTorch in here.

1 Multi-Layered Perceptrons

First, 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,

$$g(\mu(x)) = x^T\beta = \eta(x)$$

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.

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

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 $\mathcal{X}$. For example, consider the set of points in $\mathbb{R}^2$ with the corresponding class in Figure **??**. 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. There are many ways to construct parametric nonlinear basis functions and in fact, neural networks use basis functions that are of the form $\boldsymbol{\phi}(\mathbf{x}) = g^{-1}(\mathbf{x}^T\boldsymbol{\beta})$.

A neuron basically takes in a vector $\mathbf{x} \in \mathbb{R}^d$ and multiplies its corresponding weight by some vector $\boldsymbol{\omega}$, plus some bias term b. It is then sent into some nonlinear activation function $\sigma : \mathbb{R} \longrightarrow \mathbb{R}$. Letting the parameter be $\theta = (\boldsymbol{\omega}, b)$, we can think of a neuron as a function

$$h_\theta(\mathbf{x}) = f(\boldsymbol{\omega}^T\mathbf{x} + b)$$

A single neuron with the activation function as the step function

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

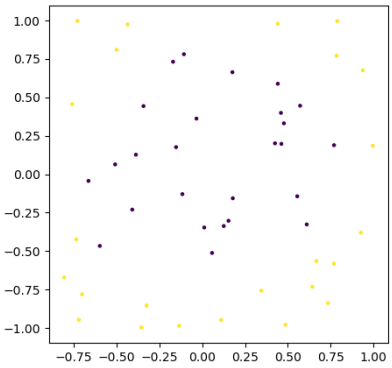
is simply the perceptron algorithm. It divides $\mathbb{R}^d$ using a hyperplane $\boldsymbol{\omega}^T\mathbf{x} + b = 0$ and linearly classifies all points on one side to value 1 and the other side to value 0. This is similar to a neuron, which takes in a value and outputs a "signal" if the function evaluated gets past a threshold. However, 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 (visualized as a node) that takes in inputs $\mathbf{x}$ and outputs a value y calculated

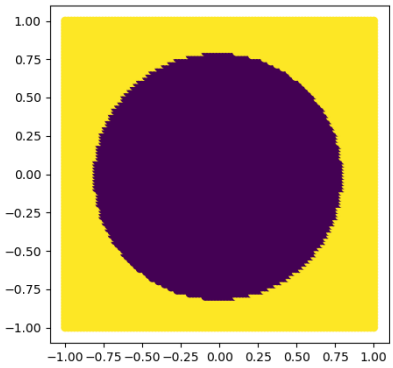
$$y = \sigma(\mathbf{w}^T x + b)$$

where σ is an activation function. Activation functions are usually simple functions with a range of $[0, 1]$ or $[-1, 1]$, and popular ones include:

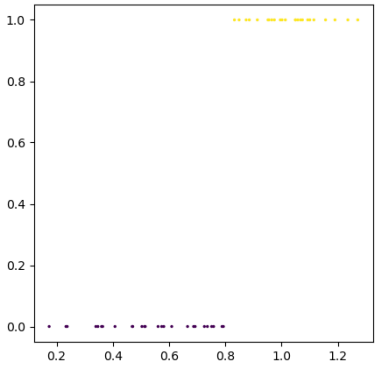
1. the rectified linear unit
$$\text{ReLU}(z) = \max\{0, z\}$$
2. the sigmoid
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$



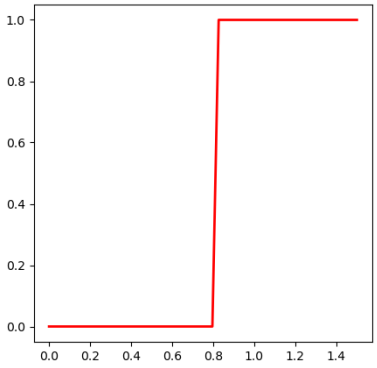
(a) Data in space $\mathcal{X} = \mathbb{R}^2$.



(b) Logistic fit to data in input space.



(c) Transformed data $\phi(\mathbf{x}) = ||\mathbf{x}||$.



(d) Logistic fit in transformed space.

Figure 1: A nonlinear feature transformation ϕ will cause a nonlinear decision boundary when doing logistic regression.

3. the hyperbolic tangent

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

A visual of a neuron is shown in Figure 2.

If there does not exist any arrow from a potential input $\mathbf{x}$ to an output y, then this means that $\mathbf{x}$ 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. What we have seen just now is a 1-layer neural network.

Definition 1.2 (Multilayer Perceptron)**.** A L-layer MLP $\mathbf{h}_\theta : \mathbb{R}^D \longrightarrow \mathbb{R}^M$ is the function

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

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

1. The inputs will be labeled $\mathbf{x} = \mathbf{a}^{[0]}$ which is in $\mathbb{R}^{N^{[0]}} = \mathbb{R}^D$.

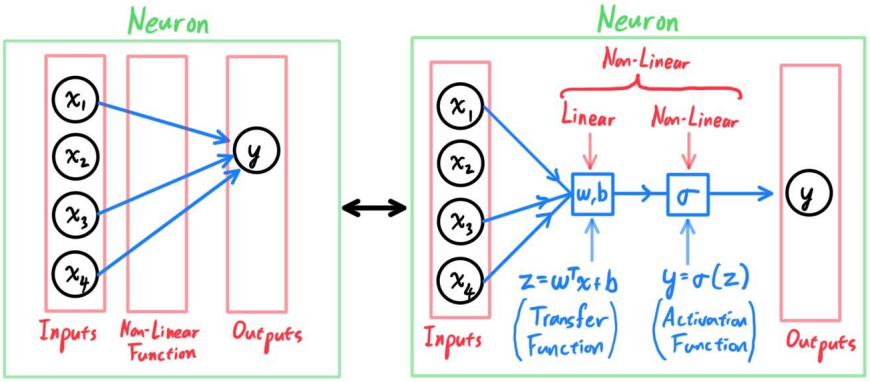
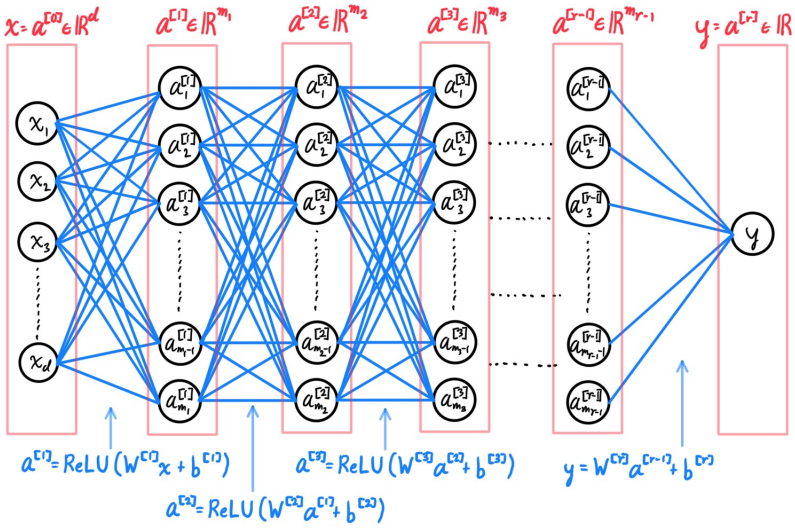


Figure 2: Diagram of a neuron, decomposed into its linear and nonlinear components.

2. We map $\mathbf{a}^{[l]} \in \mathbb{R}^{N^{[l]}} \mapsto \mathbf{W}^{[l+1]}\mathbf{a}^{[l]} + \mathbf{b}^{[l+1]} = \mathbf{z}^{[l+1]} \in \mathbb{R}^{N^{[l+1]}}$, where z denotes a vector after an affine transformation.
3. We map $\mathbf{z}^{[l+1]} \in \mathbb{R}^{N^{[l+1]}} \mapsto \boldsymbol{\sigma}(\mathbf{z}^{[l+1]}) = \mathbf{a}^{[l+1]} \in \mathbb{R}^{N^{[l+1]}}$, where a denotes a vector after an activation function.
4. We keep doing this until we reach the second last layer with vector $\mathbf{a}^{[L-1]}$.
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 $\mathbf{a}^{[L-1]} \mapsto \mathbf{W}^{[L]}\mathbf{a}^{[L-1]} + \mathbf{b}^{[L]} = \mathbf{z}^{[L]}$, followed by the link function to get our prediction: $\mathbf{a}^{[L]} = \mathbf{g}(\mathbf{z}^{[L]}) = \mathbf{h}_{\boldsymbol{\theta}}(\mathbf{x}) \in \mathbb{R}^M$.

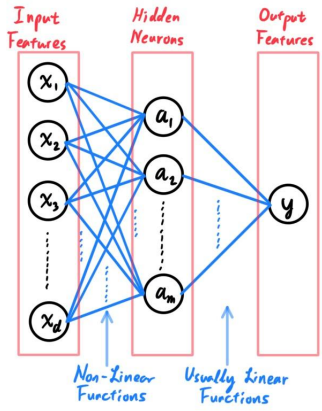
All the parameters of the neural net will be denoted $\boldsymbol{\theta}$. Ultimately, a neural net is really just a generalized linear model on an feature space with a ton of nonlinear preprocessing.



In reality, all these processes are done using minibatches, so given a minibatch of size R, our input $\mathbf{x} \in \mathbb{R}^{D \times R}$.

The last layer represents the key features that we are interested in, and 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.

Example 1.1. The **fully-connected 2-layer neural network** of d input features $\mathbf{x} \in \mathbb{R}^d$ and one scalar output $y \in \mathbb{R}$ can be visualized below. It has one **hidden layer** with m inputs values $a_1, \dots, a_m$.



Conventionally, we account for every layer except for the final layer when talking about the number of layers in the neural net.

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.

1.1 Data Preprocessing

Data preprocessing is similar to regular supervised learning models. We standardize the data so that all the features are weighted equally. We can also use PCA or diagonalize the covariates.

1.2 Weight Initialization

Now how should we initialize our weights?

1. If we set $\boldsymbol{\theta} = \mathbf{0}$, i.e. set all weights to 0, then all of 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.
2. Therefore, the next thing to do is initialize all weights according to small independent Gaussians $N(0, 0.1)$. However, this has problems since for many layer networks, as we multiply our inputs by small values over and over, we will eventually converge to a vector of 0s for each hidden layer. The activations will go to 0 and the gradients also 0, and so there will be no learning.
3. If we initialize them as random weights with a large norm, then the activations may saturate (for tanh), meaning that the gradients will be 0 and there will be no learning.

Therefore, we can use something called **Xavier initialization** or **He initialization**. Proper initialization is an active area of research, and PyTorch will automatically implement the latest initialization for us in our layer constructors.

1.3 Weight Space Symmetries

1.4 Backpropagation

Now we can essentially do the same regression analysis with neural nets. Assume that we have some neural net $h_{\boldsymbol{\theta}}$, and denote the set of all functions of this form to be $\mathcal{F} = \{h_{\boldsymbol{\theta}} : \boldsymbol{\theta} \in \mathbb{R}^M\}$, where M is the number of parameters in this model. Then, we will assume that $h_{\boldsymbol{\theta}}(X)$ approximates $\mathbb{E}[Y \mid X]$ in such a way that

$$Y = h_{\boldsymbol{\theta}}(X) + \epsilon, \quad \epsilon \sim N(0, \sigma^2)$$

Then the distibution of $Y \mid X = x$ would have density

$$p(y \mid \mathbf{x}, \boldsymbol{\theta}) = N(y \mid h_{\boldsymbol{\theta}}(\mathbf{x}), \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{\big(y - h_{\boldsymbol{\theta}}(\mathbf{x})\big)^2}{2\sigma^2} \right)$$

and taking the log likelihood of the dataset $\{(\mathbf{x}^{(n)}, y^{(n)})\}_{n=1}^N$ gives us

$$\ell(\boldsymbol{\theta}) = \frac{1}{2\sigma^2} \sum_{n=1}^N \big(y - h_{\boldsymbol{\theta}}(\mathbf{x})\big)^2 + \frac{N}{2} \ln \sigma^2 + \frac{N}{2} \ln(2\pi)$$

which clearly shows that we must minimize our sum of squares error.

Now remember that our neural net is really just a generalized linear model where we are learning the transformations in addition to the final parameter weights. Therefore, our outputs y may not be in $\mathbb{R}$ (such as softmax activation), which will give different loss functions, and so we should generalize our loss to be

$$E(\boldsymbol{\theta}) = \sum_{n=1}^N E_n\big[\mathbf{y}^{(n)}, h_{\boldsymbol{\theta}}(\mathbf{x}^{(n)})\big] = \sum_{n=1}^N E_n(\boldsymbol{\theta})$$

where E_n is the loss corresponding to the nth input-output pair.

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 $\partial E / \partial \boldsymbol{\theta}$.

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.

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

It helps to visualize that we are focusing on

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

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

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

$w_{ji}^{[l]}$ 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

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

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

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

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

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

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

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

which can be rewritten in the matrix notation

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

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, $N^{[L]} = 1$).

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

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

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

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

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

   which gives the cross entropy error

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

   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

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

giving us

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

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

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

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

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

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**:

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

which gives the matrix form

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

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

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

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.

1.4.1 Summary

Therefore, let us summarize what a MLP does:

1. *Initialization*: We initialize all the parameters to be

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

2. *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.

3. *Forward Propagation*: Apply input vector $\mathbf{x}^{(n)}$ and use forward propagation to compute the values of all the hidden and activation units

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

4. *Back Propagation*:

   (a) Evaluate the $\boldsymbol{\delta}^{[l]}$'s starting from the back with the formula

$$\begin{aligned} \boldsymbol{\delta}^{[L]} &= \left(\frac{\partial \mathbf{g}}{\partial \mathbf{z}^{[L]}}\right)^T \left(\frac{\partial E_n}{\partial \mathbf{a}^{[L]}}\right) \\ \boldsymbol{\delta}^{[l]} &= \boldsymbol{\sigma}'(\mathbf{z}^{[l]}) \odot (\mathbf{W}^{[l+1]})^T \boldsymbol{\delta}^{[l+1]} \quad l = 1, \dots, L-1 \end{aligned}$$

   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

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

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

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

   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.

1.5 Neural Net from Scratch

Now let us implement a neural network with batch gradient descent in Python from scratch, using only Numpy. We will train on the MNIST dataset where the train and test sets can be gotten using the following commands.

```
import numpy as np
import torchvision.datasets as datasets

train_set = datasets.MNIST('./data', train=True, download=True)
test_set = datasets.MNIST('./data', train=False, download=True)

# Check the lengths of train sets and test sets
assert len(train_set) == 60000 and len(test_set) == 10000
```

Now we want to take each 28×28 image and flatten it out to a 784 vector.

```
X_train = np.array([picture.numpy().reshape(-1) for picture in train_set.data]).T / 255.
Y_train = train_set.targets.numpy()
X_test = np.array([picture.numpy().reshape(-1) for picture in test_set.data]).T / 255.
Y_test = test_set.targets.numpy()

# Check shapes
assert X_train.shape == (784, 60000) and Y_train.shape == (60000, )
```

Here are some helper functions that we will need.

```
def initialize_params():
    W1 = np.random.uniform(-1, 1, size=(10, 784))
    b1 = np.random.uniform(-1, 1, size=(10, 1))
    W2 = np.random.uniform(-1, 1, size=(10, 10))
    b2 = np.random.uniform(-1, 1, size=(10, 1))
    return W1, b1, W2, b2

def oneHot(Y):
    # Y is 60000
    oneHotY = np.zeros((10, Y.size))
    oneHotY[Y, np.arange(Y.size)] = 1
    return oneHotY # 10x60000

def ReLU(Z):
    return np.maximum(0, Z)

def ReLU_d(Z):
    return Z > 0

def softMax(X:np.array):
    x_max = np.max(X, axis=0)
    X = X - x_max
    return np.exp(X) / np.sum(np.exp(X), axis=0)

def softMax_d(X:np.array):
    sm = softMax(X)
    return - (np.diag(sm.sum(axis=1)) - np.matmul(sm, np.transpose(sm))) / X.shape[1]
```

Now we implement the forward propagation and back propagation.

```
def forwardProp(W1, b1, W2, b2, X):
    # Z1 12x1, W1 12x784 , X 784x1, b1 12x1
    Z1 = np.matmul(W1, X) + b1
    # A1 12x60000
    A1 = ReLU(Z1)

    # Z2 10x1, W2 10x12, A1 12x60000, b2 10x60000
    Z2 = np.matmul(W2, A1) + b2
    # A2 10x60000
    A2 = softMax(Z2)
    return Z1, A1, Z2, A2

def backProp(Z1, A1, Z2, A2, W1, W2, X, Y):
    N = Y.size
    oneHotY = oneHot(Y)

    # 10x1 = 10x10 10x1
    error2 = A2 - oneHotY
    # error2 = np.matmul(np.transpose(softMax_d(Z2)), A2 - oneHotY)
    # 10x12 = 10x1 1x12
    dW2 = 1/N * np.matmul(error2, A1.T)
    # 10x1
    dB2 = 1/N * error2.sum(axis=1)

    # 12x1 = 12x1 .* 12x10 10x1
    error1 = np.vectorize(ReLU_d)(Z1) * np.matmul(W2.T, error2)
    # 12x784 = 12x1 1x784
    dW1 = 1/N * np.matmul(error1, X.T)
    # 12x1
    dB1 = 1/N * error1.sum(axis=1)

    return dW1, dB1, dW2, dB2
```

We now build the batch gradient descent algorithm.

```
def get_predictions(A2):
    return np.argmax(A2, 0)

def get_accuracy(predictions, Y):
    print(predictions, Y)
    return np.sum(predictions == Y) / Y.size

def update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha):
    W1 = W1 - alpha * dW1
    b1 = b1 - alpha * db1.reshape(-1, 1)
    W2 = W2 - alpha * dW2
    b2 = b2 - alpha * db2.reshape(-1, 1)
    return W1, b1, W2, b2

def gradient_descent(X, Y, alpha, iterations):
    W1, b1, W2, b2 = initialize_params()
    for i in range(iterations):
        Z1, A1, Z2, A2 = forwardProp(W1, b1, W2, b2, X)
        dW1, db1, dW2, db2 = backProp(Z1, A1, Z2, A2, W1, W2, X, Y)
        W1, b1, W2, b2 = update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha)
        if i % 10 == 0:
            print("Iteration: ", i)
            predictions = get_predictions(A2)
            print(get_accuracy(predictions, Y))
    return W1, b1, W2, b2

# Run it
W1, b1, W2, b2 = gradient_descent(X_train, Y_train, 0.1, 500)
```

Which gives the following output, ultimately yielding a 77% accuracy rate in detecting handwritten digits.

```
Iteration:  0
[4 7 4 ... 4 7 0] [5 0 4 ... 5 6 8]
0.10175
Iteration:  10
[8 0 4 ... 4 6 0] [5 0 4 ... 5 6 8]
0.20715
Iteration:  20
[8 0 4 ... 4 6 0] [5 0 4 ... 5 6 8]
0.2333
Iteration:  30
[9 0 4 ... 4 6 0] [5 0 4 ... 5 6 8]
0.2579166666666667
...
...
...
Iteration:  470
[3 0 9 ... 5 6 0] [5 0 4 ... 5 6 8]
0.7645
Iteration:  480
[3 0 9 ... 5 6 0] [5 0 4 ... 5 6 8]
0.7646833333333334
Iteration:  490
[3 0 9 ... 5 6 0] [5 0 4 ... 5 6 8]
0.7728166666666667
```

1.6 Quick Start to PyTorch

There is no more of a reason to go any further with vanilla Python. We will use the PyTorch package, which manipulates `torch.tensor` objects similar to `numpy.array` objects. We will not go over the basic tensor object here. Now in neural nets, most of the training algorithms are basically matrix multiplication, meaning that a massively parallelized architecture is best. Therefore, we would like to run our modules in the `cuda` device.

```
import torch

device = (
    "cuda"
    if torch.cuda.is_available()
    else "mps"
    if torch.backends.mps.is_available()
    else "cpu"
)
print(f"Using {device} device")
```

1.6.1 Datasets and Features/Label Transformations

Now we have popular datasets for machine learning. It is good to know what they are, what the input and output data consists of, and its size.

1. *MNIST* consists of 70k (60k training + 10k test) 28×28 grayscale images in 10 classes $(0, 1, \dots, 9)$. It has a bunch of handwritten digits. 50MB
2. *Fashion-MNIST* consists of 70k (60k + 10k) 28×28 grayscale images in 10 classes (top, trouser, pullover, dress, coat, sandal, shirt, sneaker, bag, ankle boot).
3. *CIFAR-10* consists of 60k (50k train + 10k test) 32×32 color images in 10 classes (airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck). 170MB.
4. *CIFAR-100* is just like the CIFAR-10, but with 100 classes containing 600 images each. CIFAR-10 is a standard benchmark for most classification tasks, while CIFAR-100 provides a more challening classification problem.
5. *ImageNet* consists of 1.33m (1.28m + 50k) color images in 1000 classes, with a variety of resolutions, but many researchers crop/compress them down to 224×224. 150GB

Now we can load these datasets with the following command. If they are not found in the `root` directory specified in the parameters, then they will be downloaded. Since we may be working with different datasets, we should have a `data` folder containing multiple subfolders for each dataset.

```
import os
import torch
from torchvision import datasets
from torchvision.transforms import ToTensor

training_data = datasets.FashionMNIST(
    root="data",            # the folder where the data will be stored in
    train=True,             # training or test dataset
    download=True,          # downloads data if not available at root
    transform=ToTensor()    # specifies features/label transformations
)
test_data = datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor()
)
```

Now the `training_data` and `test_data` are lists of 2-tuples of the input image (tensor object of size 1, 28, 28) and the output label (integer).

```
img, label = training_data[0]
print(img.shape)        torch.size([1, 28, 28])
print(label)            9
```

We can manually map the integers to the category word, and use the `torch.squeeze` method to get rid of the extra dimension of 1, before plotting the image, which is shown by Figure 3.

```
labels_map = {
    0: "T-Shirt",
    1: "Trouser",
    2: "Pullover",
    3: "Dress",
    4: "Coat",
    5: "Sandal",
    6: "Shirt",
    7: "Sneaker",
    8: "Bag",
    9: "Ankle Boot",
}

random_index = torch.randint(len(training_data), size=(1,)).item()
img, label = training_data[random_index]

# Plot
plt.title(f"{labels_map[label]}")
plt.imshow(torch.squeeze(img), cmap="gray")
plt.axis("off")
```

Now that we have loaded our dataset, we can retrieve our features and labels one sample at a time. However, while training a model, we typically want to pass the samples in minibatches (e.g. in SGD), reshuffle the data at every epoch to reduce model overfitting, and use Python's multiprocessing to speed up data retrieval. We can do all this with the `DataLoader` API.

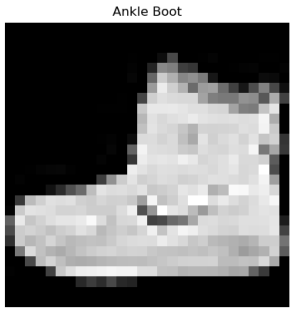


Figure 3: Data point from FashionMNIST

```
from torch.utils.data import DataLoader

train_dataloader = DataLoader(training_data,    # our dataset
                              batch_size=64,    # batch size
                              shuffle=True      # shuffling the data
                             )
test_dataloader = DataLoader(test_data, batch_size=64, shuffle=True)

train_features, train_labels = next(iter(train_dataloader))
print(f"Feature batch shape: {train_features.size()}")
print(f"Labels batch shape: {train_labels.size()}")

# Feature batch shape: torch.Size([64, 1, 28, 28])
# Labels batch shape: torch.Size([64])
```

Like in machine learning, we would like to normalize our data in some way. Fortunately, when loading the data, this is automatically done for us. The Fashion image are in PIL format, and the labels are just integers.

```
from torchvision.transforms import ToTensor, Lambda

ds = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor(),
    target_transform=Lambda(lambda y: torch.zeros(10, dtype=torch.float).scatter_(0,
        torch.tensor(y), value=1))
)
```

1. `transform=ToTensor()` tells us to take the images, convert them to a tensor, covert all the elements to floats, and normalize them.
2. `target_transform` tells us to one-hot encode the integer labels to vectors. It first creates a zero tensor of size 10 (the number of labels in our dataset) and calls `scatter_` which assigns a `value=1` on the index as given by the label y.

1.6.2 Building a Neural Net

We can build a neural net as a subclass of `nn.Module`. Note that the `nn.flatten` method flattens the tensor with `start_dim` set to 1 by default to avoid flattening the first axis (usually the batch axis).

```
from torch import nn

class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```

All of these methods can be used separately, and they can be combined using sequential to form a composition of functions.

```
X = torch.rand(3, 28, 28)
Y = torch.rand(3, 8)

flatten = nn.Flatten()      # flattens from dim=1
linear = nn.Linear(8, 4)    # linear map
relu = nn.ReLU()            # ReLU map
softmax = nn.Softmax(dim=1) # Softmax map

print(flatten(X).size())    # torch.Size([3, 784])
print(linear(Y).size())     # torch.Size([3, 4])
print(relu(Y).size())       # torch.Size([3, 8])
print(softmax(Y).size())    # torch.Size([3, 8])
```

Next, we want to create an instance of this neural network and move it to our cuda device.

```
model = NeuralNetwork().to(device)
print(model)

# NeuralNetwork(
#   (flatten): Flatten(start_dim=1, end_dim=-1)
#   (linear_relu_stack): Sequential(
#     (0): Linear(in_features=784, out_features=512, bias=True)
#     (1): ReLU()
#     (2): Linear(in_features=512, out_features=512, bias=True)
#     (3): ReLU()
#     (4): Linear(in_features=512, out_features=10, bias=True)
#   )
# )
```

To use the model, we can just directly pass in the input data, which executes the model's forward propagation (`forward` method). Do not call `model.forward()` directly!

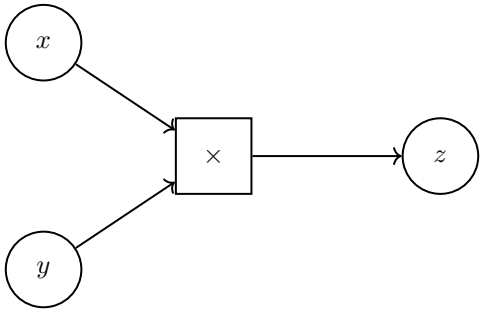
```
X = torch.rand(3, 28, 28, device=device)
logits = model(X)
pred_probab = nn.Softmax(dim=1)(logits)
y_pred = pred_probab.argmax(1)
print(f"Predicted class: {y_pred}")

# tensor([[-0.0518,  0.0379, -0.0579,  0.0050,  0.0367,  0.0958,  0.0055,  0.1385,
#           0.0464, -0.0017]], device='cuda:0', grad_fn=<AddmmBackward0>)
# tensor([[0.0924, 0.1011, 0.0919, 0.0978, 0.1010, 0.1071, 0.0979, 0.1118, 0.1019,
#          0.0972]], device='cuda:0', grad_fn=<SoftmaxBackward0>)
# Predicted class: tensor([7], device='cuda:0')
```

We can also access parameters by calling `model.named_parameters()`, which gives us a list of tuples $(name, param)$, where name is simply the name of the weight, and the param is the matrix representing the linear mapping or the bias term.

```
for name, param in model.named_parameters():
    print(f"Layer: {name} | Size: {param.size()} | Values : {param[:2]} \n")
```

1.6.3 Automatic Differentiation

The forward and backward API for a computational gate multiplying its inputs



is represented with the code below.

```
class MultiplyGate(object):
    def forward(x, y):
        z = x * y
        self.x = x      # must keep these around!
        self.y = y
        return z

    def backward(dz):
        dx = self.y * dz
        dy = self.x * dz
        return [dx, dy]
```

1.6.4 Optimizing Model Parameters

Bsaically in each epoch, we want to do two things:

1. *Train Loop*: Iterate over the (minibatch) training dataset and try to converge to optimal parameters using backprop.
2. **Test Loop**: Iterate over the test dataset to check if model performance is improving.

Once we compute the gradient of a given loss function, we can use different optimizers like SGD or ADAM to optimize.

```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(
    model.parameters(),     # which parameters to optimize
    lr=1e-3                 # learning rate
)
```

Our train loop

```
def train(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    model.train()
    for batch, (X, y) in enumerate(dataloader):
        X, y = X.to(device), y.to(device)

        # Compute prediction error
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

        if batch % 100 == 0:
            loss, current = loss.item(), (batch + 1) * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
```

We can then evaluate the model's performance against the test dataset.

```
def test(dataloader, model, loss_fn):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    model.eval()
    test_loss, correct = 0, 0
    with torch.no_grad():
        for X, y in dataloader:
            X, y = X.to(device), y.to(device)
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()
    test_loss /= num_batches
    correct /= size
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss: {test_loss:>8f} \n")
```

Now we run this through a loop over some number of epochs.

```
epochs = 5
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train(train_dataloader, model, loss_fn, optimizer)
    test(test_dataloader, model, loss_fn)
print("Done!")
```

2 Training Stability

Training stability refers to helping the training process, which does not improve the theoretical upper bound on the accuracy of the neural net, but are extremely important for practical applications. It is related to regularization, which prevents the neural net from overfitting to the data. They are both related as they improve the performance of training and the neural network itself, but it good to distinguish them. Given some loss landscape, regularization ensures that we don't "fall" into some deep hole that looks great on our training error, but does terribly on our testing error. On the other hand, training stability ensures that we update our parameters efficiently to traverse the landscape.

2.1 Optimizers

2.2 Gradient Problems

Exploding/Vanishing gradient problem.

2.3 Activation Functions

The choice of the activation function can have a significant impact on your training, and we will describe a few examples below.

Example 2.1 (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

$$f\left(\sum_i w_i x_i + b\right) \implies \frac{\partial f}{\partial w_i} = f'\left(\sum_i w_i x_i + b\right) x_i$$

   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.

Example 2.2 (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.

Example 2.3 (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.

Example 2.4 (Leaky ReLU)**.** The leaky ReLU

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

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.

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

Example 2.5 (Exponential Linear Unit)**.** 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.

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

Example 2.6 (Max-Out Neuron)**.** The maxout neuron has the following form

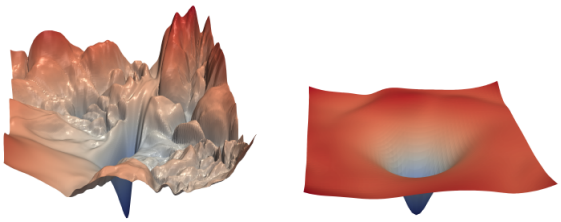
$$\sigma(\mathbf{x}) = \max\{\mathbf{w}_1^T\mathbf{x} + b_1, \mathbf{w}_2^T\mathbf{x} + b_2\}$$

This generalizes the ReLU and leaky ReLU. It is linear, which is nice, and it does not saturate, meaning that the gradient will never die. However, this doubles the number of parameters in the neuron, making it more computationally expensive.

In practice, we should do the following:

1. Use ReLU and be careful with your learning rates.
2. Try out leaky ReLU, maxout, and ELU
3. Try out tanh but don't expect much
4. Do not use sigmoid, since it is obsolete

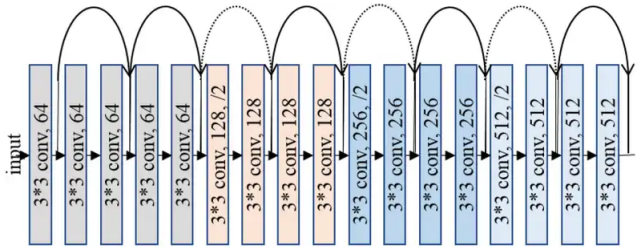
2.4 Residual Connections



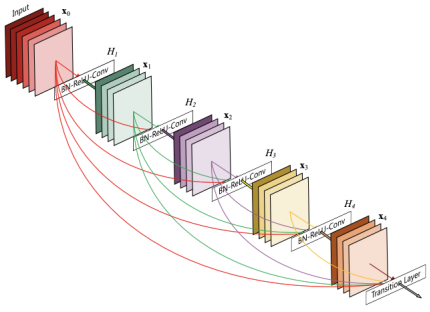
(a) without skip connections (b) with skip connections

Some residual architecture solutions include feedforward/convolutional architectures that add more direct connections from the lower layers, thus allowing the gradient to flow.

1. The **ResNet** (**Residual Connections**) includes an additional path from layer i to layer j that skips the intermediate layers and simply goes through an identity connection, which is then summed with the original path during forward prop.



2. The **DenseNet** directly connects each layer to all future layers, which allows for maximal flow but a lot of computational cost.



3. The **HighwayNet** (**Highway Connections**) is similar to residual connections, but the identity connection vs the transformation layer is controlled by a dynamic gate. These are inspired by LSTMs, but are applied to deep feedforward/convolutional networks.

2.5 Normalization Layers

Note that given the distribution of the covariates X in $\mathcal{X} \subset \mathbb{R}^D$, our neural network transforms them into different distributions in $\mathbb{R}^{[l]}$. For example, in the first layer, where both the affine and the activation functions are usually measurable, the random variable

$$\boldsymbol{\sigma}\left(\mathbf{W}^{[1]}X + \mathbf{b}^{[1]}\right)$$

will induce a probability measure over $\mathbb{R}^{[1]}$. This phenomenon where the distribution of each layer's inputs change is called **internal covariate shift**.

It has long been known that standardization of the inputs (i.e. transforming them to have 0 mean and unit variance) results in better convergence of the neural net, and extending this to hidden layers, it would be advantageous to achieve the same standardization of each layer. In one case, if our inputs to a saturated activation (like tanh, which are saturated for very positive or very negative inputs) are too big, then the gradient will die off, which is why we want to constrain them to a small interval around 0. This is called a **normalization layer**, which works like this:

1. We select a minibatch of training examples $\mathcal{B} = \{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(B)}\}$. Ideally, we would get the mean and variance over the whole batch, but in SGD, the minibatch is a good approximation of the distribution, so we take it with respec to $\mathcal{B}$.

2. We standardize the $\mathbf{x}^{(b)}$'s element-wise. That is, for a given feature dimension d,

$$\mu_d = \frac{1}{B}\sum_b x_d^{(b)}, \quad \sigma_d^2 = \frac{1}{B}\sum_b \left(x_d^{(b)} - \mu_d\right)^2$$

and we set

$$\hat{x}_d^{(b)} = \frac{x_d^{(b)} - \mu_d}{\sqrt{\sigma_d^2 + \epsilon}}, \quad \text{i.e. } \hat{\mathbf{x}}^{(b)} = (\mathbf{x}^{(b)} - \boldsymbol{\mu}) \odot (\boldsymbol{\sigma}^2 + \boldsymbol{\epsilon})^{-1/2}$$

where ϵ is a small float needed for numerical stability. This element-wise normalization is not optimal if the covariates are correlated, but in practicality this is not too big of a problem.

3. However, constraining our normalization to the unit Gaussian in many cases constrains us to the linear regime of the nonlinearity of our activation functions (e.g. sigmoid or tanh). We would like a little bit of control over how much saturation we would like to have, so we allow some rescaling and reshifting (element-wise, again) parameters for flexibility.

$$\mathbf{y} = \boldsymbol{\gamma} \odot \hat{\mathbf{x}} + \boldsymbol{\beta} = \mathrm{BN}_{\boldsymbol{\gamma},\boldsymbol{\beta}}(\mathbf{x})$$

If the network learns that $\gamma_d = \sqrt{\text{Var}(x_d)}$ and $\beta_d = \mathbb{E}[x_d]$, then it is as if there was no normalization at all, just the identity mapping. If $\mathbf{x} \in \mathbb{R}^n$, then this one normalization layer gives us $2n$ more parameters to learn in our network.

Therefore, rather than our steps being simply

$$\mathbf{z}^{[l]} \mapsto \boldsymbol{\sigma}(\mathbf{z}^{[l]}) = \mathbf{a}^{[l]}$$

we now have the normalization later

$$\mathbf{z}^{[l]} \mapsto \boldsymbol{\sigma}\big(\text{BN}_{\boldsymbol{\gamma},\boldsymbol{\beta}}(\mathbf{z}^{[l]})\big) = \mathbf{a}^{[l]}$$

In test time, we have the trained parameters, along with the $\boldsymbol{\gamma}$'s and $\boldsymbol{\beta}$'s, but we now recompute the normalization constants with respect to the entire training set, not a minibatch.

2.5.1 Max Norm Regularization

Though large momentum and learning rate speed up learning, they sometimes cause the network weights to grow very large. To prevent this, we can use max-norm regularization. This constraints the norm of the vector of incoming weights at each hidden unit to be bound by a constant c. Typical values of c range from 3 to 4.

3 Regularization

3.1 Early Stopping

3.2 L1 and L2 Regularization

3.3 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 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.

In PyTorch, this can be done with the dropout function, where `p` represents the probability of an element getting dropped out and `inplace=True` means that the operation will be done in place (i.e. the variable itself will be changed). We should set it to false since we don't want to set all the parameters to 0 permanently.

```
drop = nn.Dropout(p=0.5, inplace=False)
X = torch.rand(4)
print(X)            # tensor([0.3342, 0.8208, 0.3765, 0.1820])
print(drop(X))      # tensor([0.6684, 1.6417, 0.0000, 0.0000])
print(X)            # tensor([0.3342, 0.8208, 0.3765, 0.1820])
```

```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Dropout(p=0.5, inplace=False),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Dropout(p=0.5, inplace=False),
            nn.Linear(512, 10)
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```

When you call `model.eval()`, PyTorch automatically turns off Dropout (as well as other functionalities such as BatchNorm). Similarly, when you call `model.train()`, Dropout is turned on and will be used during training. It is expected that a dropout network with M hidden units in each layer will have pM units after dropout. Therefore, if an M-sized layer is optimal for a standard neural net on any given task, then a good dropout net should have at least M/p units.

3.4 Network Pruning

3.5 Data Augmentation

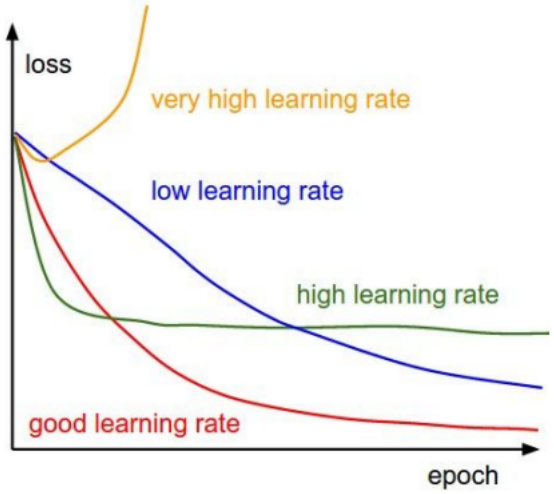
It is well known that having more training data helps with overfitting, and so we may be able to perform basic transformations to our current data to artificially generate more training data. For example, if we have images, then we can flip, crop, translate, rotate, stretch, shear, and lens-distort these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

3.6 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
$$\frac{||\nabla\boldsymbol{\theta}||}{||\boldsymbol{\theta}||} \approx 0.001 \text{ or } 0.01$$

4 Convolutional Neural Networks

4.1 Kernels

A convolution is described by a **kernel**, also called a **filter**, which is simply a $K \times K$ matrix. It does not have to be square but is conventionally so. It goes through a grayscale image at every point and compute the dot product of the kernel with the overlapping portion of the image, creating a new pixel. This can be shown in Figure 4.

Now if this was a color image, then the $K \times K$ kernel $\mathcal{K}$ would dot over all 3 layers, without changing over all 3 layers. This is equivalent to applying the kernel over all 3 channels separately, and then combining them together into one. 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. To construct our custom kernel, we can simply create a custom matrix:

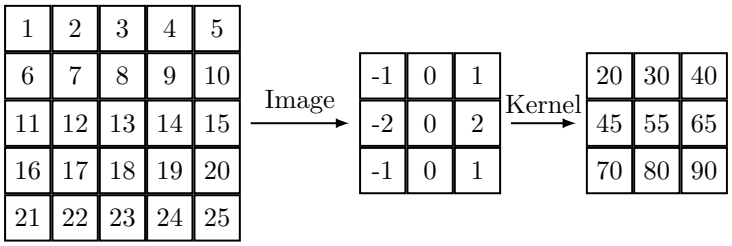


Figure 4: Convolution using a kernel on an image.

```
img = cv2.imread("cats.jpg")

# create custom 5x5 kernel
kernel = (1/25) * np.ones((5, 5), dtype=np.float32)

# apply to image
dst = cv2.filter2D(img, -1, kernel)
cv2.imshow("Park", dst)
cv2.waitKey(0)
```

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.

Example 4.1 (Mean Blur, Gaussian Blur)**.** The mean and Gaussian blur is defined with kernels that are distributed uniformly and normally across the entire matrix. You can see how this would blur an image since for every pixel, we take the weighted average over all of its surrounding pixels.

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

On a large scale, there really aren't any discernable differences, as seen in Figure **??**, but the Guassian blur is known to be a more realistic representation of how humans receive blur.

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

Figure 5: Comparison of blurring kernels on image.

Example 4.2 (Sharpening)**.** A sharpening of an image would be the opposite of a blur, meaning that we emphasize the center pixel and reduce the surrounding pixels.

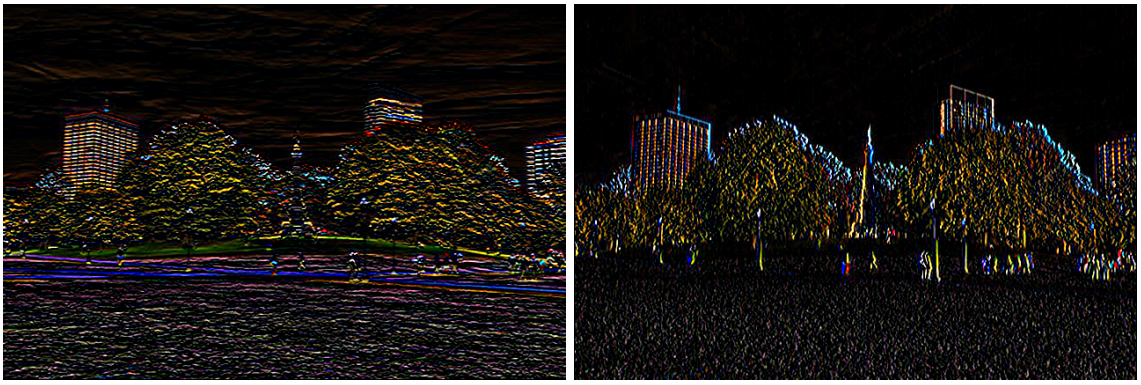
$$\text{Sharpen} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

(a) Original image. (b) 3×3 sharpening applied.

Figure 6: Sharpening kernels applied to image.

Example 4.3 (Edge Detection)**.** The edge detecting kernel looks like the following, which differs for horizontal and vertical edge detection. Note that the sum of all of its values equal 0, which means that for areas that have a relatively constant value of pixels, all the surrounding ones will "cancel" out and the kernel will output a value of 0, corresponding to black. This is why we see mostly black in the photo.

$$\text{Horizontal} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \qquad \text{Vertical} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

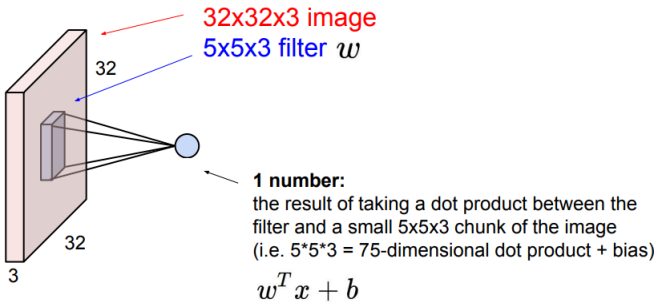


(a) 3×3 horizontal edge detecting kernel applied. (b) 3×3 vertical edge detecting kernel applied.

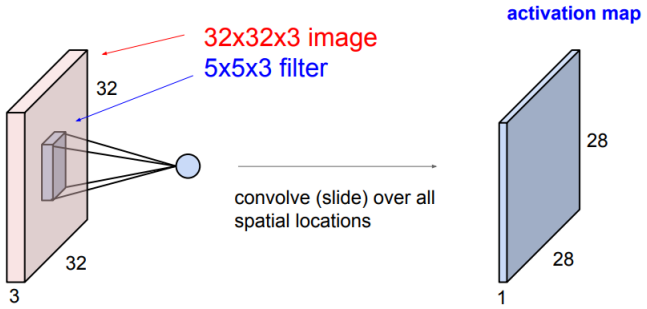
Figure 7: Edge detecting kernels applied to image.

4.2 Convolutional Layers

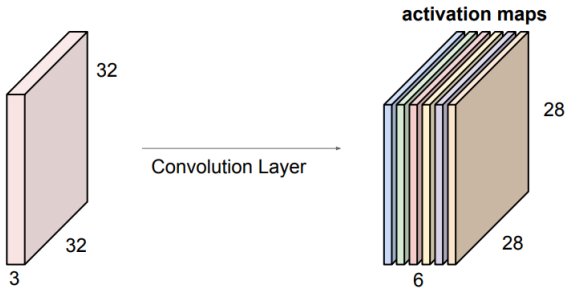
Given an image of dimension $W \times H \times D$ (where D is the depth, or number of color channels), we can take a convolution over this image by multiplying the matrix $\mathcal{K} \in \mathbb{R}^{K \times K}$ over each depth channel. Before, we have said that $\mathcal{K}$ is applied to each color channel layer, so the total number of parameters that govern this convolution is K^2. For greater flexibility, we would like our kernel to be of form $\mathcal{K} = \mathbb{R}^{K \times K \times D}$, where the kernel applied in each color channel could be different. Furthermore, we would like a bias term $b \in \mathbb{R}$, giving us a total of $DK^2 + 1$ parameters to optimize over to get the "best possible" convolution.



It is almost always the case that $D = 3$, so we will keep it to $3K^2 + 1$. A possible hyperparameter that needs to be set is the value of K, determining the **width** of the kernel. Another hyperparameter is the **stride** of the kernel, which determines how many pixels you want the filter to move. The filter will start on the top left of the image and stride (at a certain rate) to the right and down until it reaches the bottom right. Once finished, we should have a transformed image, which we can invoke our activation function $\boldsymbol{\sigma}$ on for each element (pixel) to introduce some nonlinearity.



If one filter $\mathcal{K}$ gives us one output image, then a collection of filters $\mathcal{K}_1, \dots, \mathcal{K}_l$ (almost always assumed to be of the same size) will give us a collection of l images, or also known as one image of depth l.



These convolutions help extract some sort of information, and our choice of convolutions (i.e. the $3K^2$ numbers) will extract different information. Therefore, this is called a **convolution layer**. As we have more and more convolution layers, we are able to extract from low-level, to mid-level, to high-level features in an image that we can ultimately train on. Ultimately, we would like to stack these layers together enough so that we can have our features, and then we run a few layers of MLP to get our prediction.

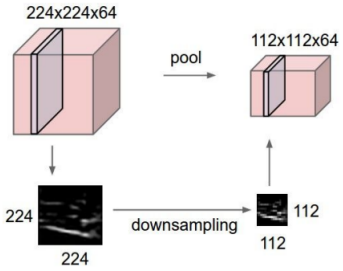
4.2.1 Convolutions as Sparase Matrix Multiplication

In a way, this is really just a giant sparse matrix multiplication since we can stretch the image out, and each convolution on a pixel is just a linear combination of the pixels around it (plus the ones around it in other channels). In a vanilla MLP, you would optimize all the parameters possible of a linear map over the input images, but a convolutional one takes a very small subset of parameters to optimize, setting everything else to 0. The weights are furthermore shared/reused across different rows called **weight sharing**, allowing the model to learn shared representations. There are three reasons that we do this:

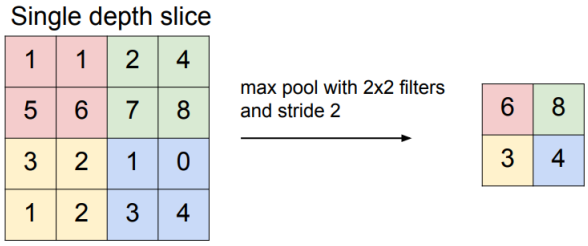
1. If the dimensions of the input image was D_1 and that of the output image was D_2, then we would have to optimize a matrix with $D_1 \times D_2$ parameters. Note that the smallest images in today's standards are $3 \times 256 \times 256 \approx 200,000$ pixels, and so $D_1 \times D_2 \approx 4 \cdot 10^{10}$, which is too many even for one layer.
2. There is an MLP way of doing computer vision, and we have performed MLP on very small grayscale images like the MNIST with decent results. Though MLP is more generalized, CNNs have comparable performance at a fraction of the computational cost.
3. The images are spatially ordered data. That is, the order in which the pixels are arranged to form the matrix provides a great help in features extraction, so this sense of locality in convolutions help.

4.3 Pooling Layers

Eventually, we would like to use convolutional layers to extract perhaps a few hundred or thousand features that we can then run a MLP on for the last few layers, followed by whatever final activation function needed for prediction. The problem is the high-dimensionality of our inputs. Recall that even the smallest images are $3 \times 256 \times 256 \approx 200,000$ pixels. Even after multiple convolutional layers, the dimensions may not decrease as fast as we want, so we introduce **pooling layers**, which are efficient layers that decrease the dimension of its inputs in a controlled way, called **downsampling**.



We can think of this as decreasing the resolution of the image, and the most common way is through **max pooling**. You basically have a $P \times P$ square window with some stride, and for each stride, we take the largest value in the window.

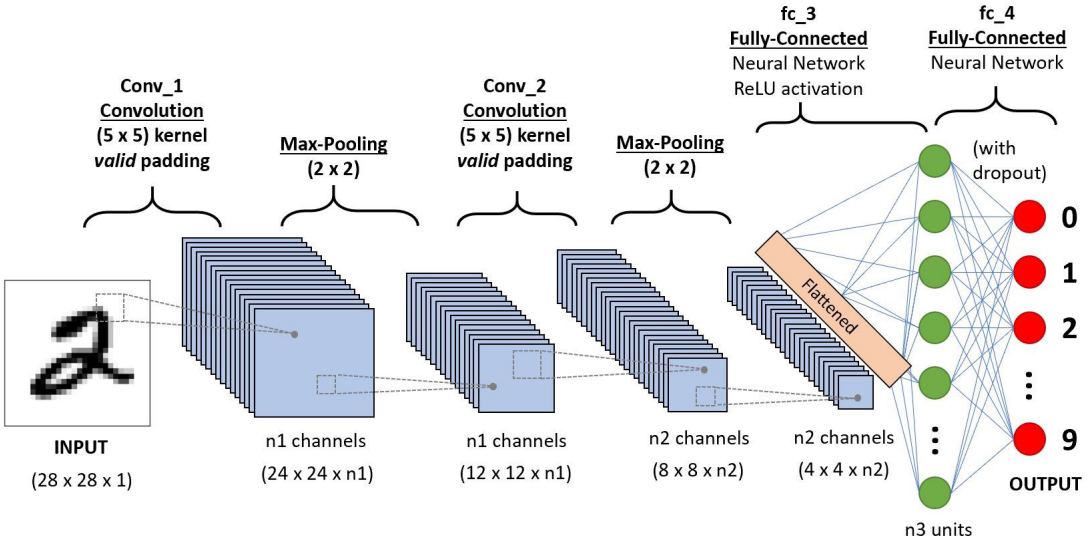


4.4 Backpropagation

4.5 Implementation from Scratch

4.6 Total Architecture with PyTorch

We have a bunch of convolutional layers with RELU, along with some pooling layers in between. Once we've done enough pooling, we just stretch the resulting image out and run a MLP on the rest with a softmax link function.



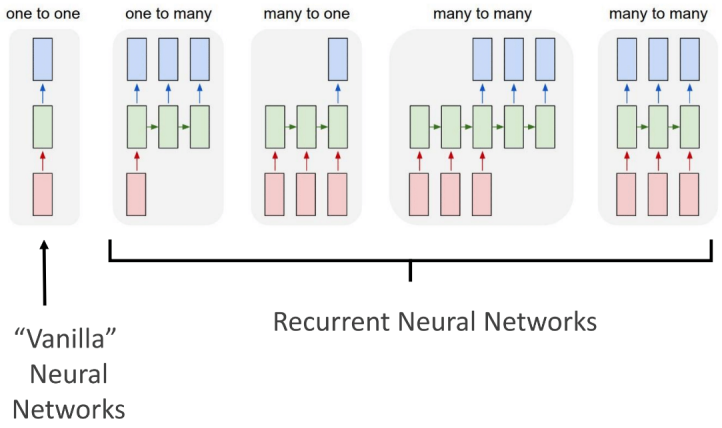
We can implement it directly in PyTorch by calling `nn.Conv2d()`, where the parameters are number of input channels, number of output channels, and size of kernel (along with optional stride, padding, bias parameters), and `nn.MaxPool2d()`, which take in the kernel size and stride. The rest of the code is completely the same.

```
class NeuralNetwork(nn.Module):
    # Convolutional Neural Network

    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)


    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

5 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 $\mathbf{g}$.

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

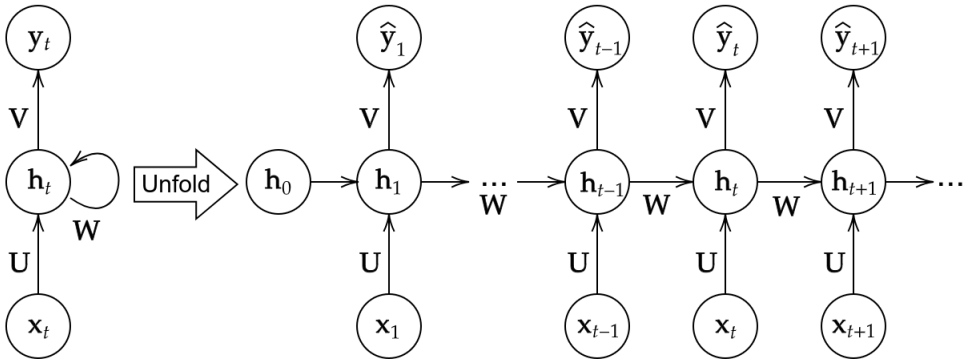
The fact that we can factorize $\mathbf{g}_t$ into a repeated application of function $\mathbf{f}$ gives us two advantages:

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.

5.1 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



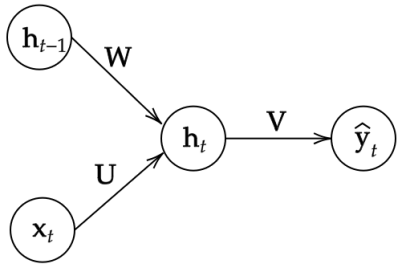
The activation functions that map to the hidden nodes and the outputs will be labeled $\boldsymbol{\sigma}_h$ and $\boldsymbol{\sigma}_y$, respectively. In general the W will represent the left and right mappings between hidden nodes, the U will represent the map going up from the input or hidden node 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.

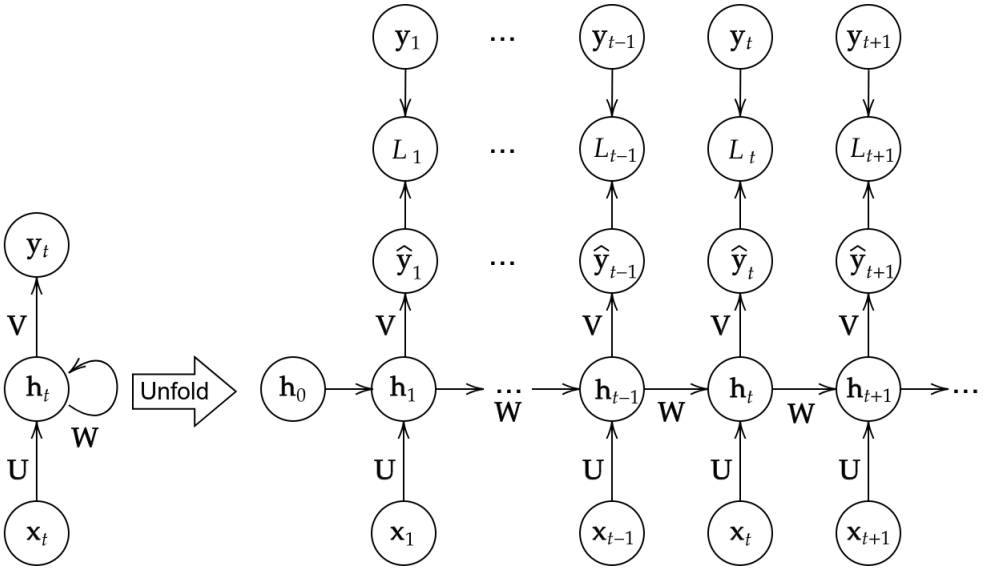


5.1.1 Loss Functions

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.

5.1.2 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:

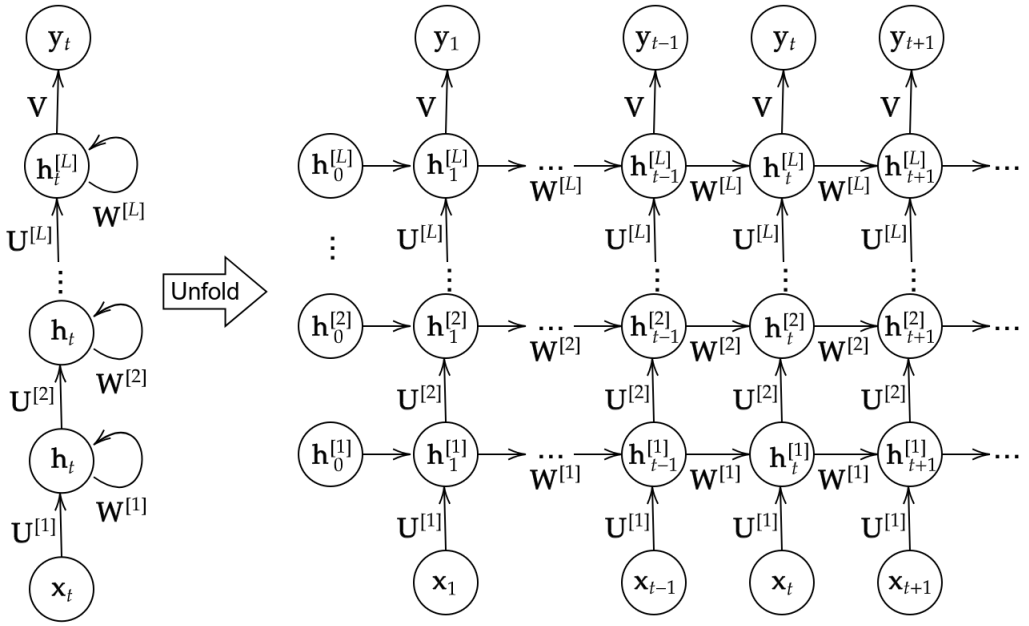
$$\sum_{k=1}^{t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} \frac{\partial \mathbf{h}_k}{\partial \mathbf{W}} = \sum_{k=1}^{t} \prod_{k<i\leq t} \frac{\partial \mathbf{h}_i}{\partial \mathbf{h}_{i-1}} \frac{\partial \mathbf{h}_k}{\partial \mathbf{W}}$$

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 $O(\tau)$.

5.1.3 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 $\mathbf{U}^{[1]} \in \mathbb{R}^{h \times d}$.

5.2 Bidirectional RNNs

5.2.1 PyTorch Implementation

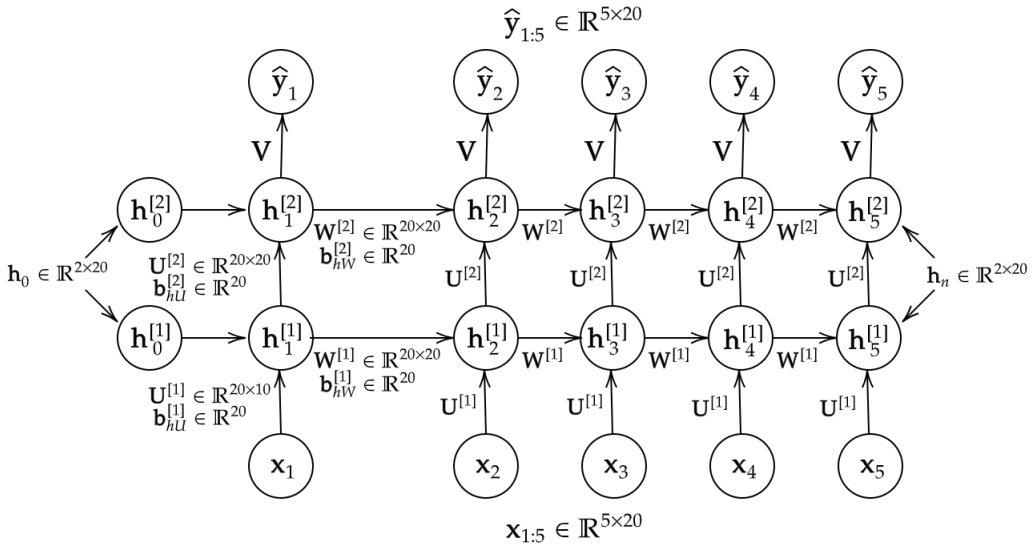
The implementation in PyTorch actually uses *two* bias terms $\mathbf{b}_{hW}^{[l]}$ and $\mathbf{b}_{hU}^{[l]}$ rather than just $\mathbf{b}_h^{[l]}$. This is technically not needed since the bias terms will just cancel out, but this is just how cuDNN (Cuda Deep Neural Network) is implemented.

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

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: $\mathbf{W}^{[1]}, \mathbf{W}^{[2]}, \mathbf{U}^{[1]}, \mathbf{U}^{[2]}, \mathbf{b}_{hU}^{[1]}, \mathbf{b}_{hW}^{[1]}, \mathbf{b}_{hW}^{[2]}, \mathbf{b}_{hU}^{[2]}$.

5.3 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.

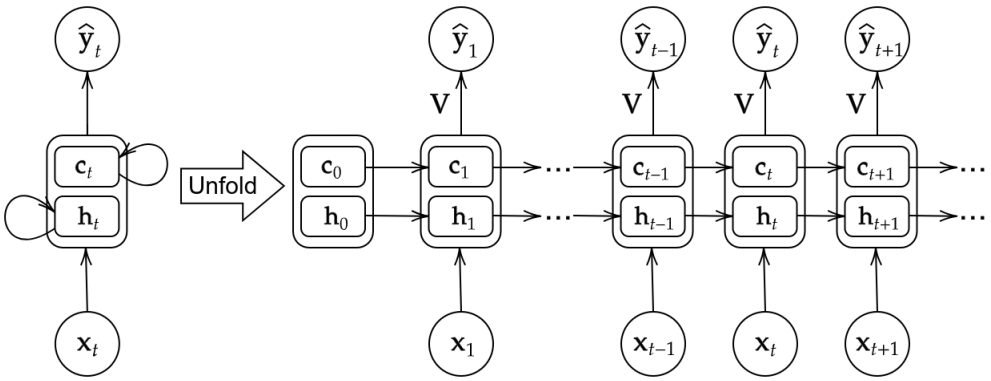
3. The **LSTM** model uses a memory cell for modeling long-range dependencies and avoids the vanishing gradient problems.

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

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

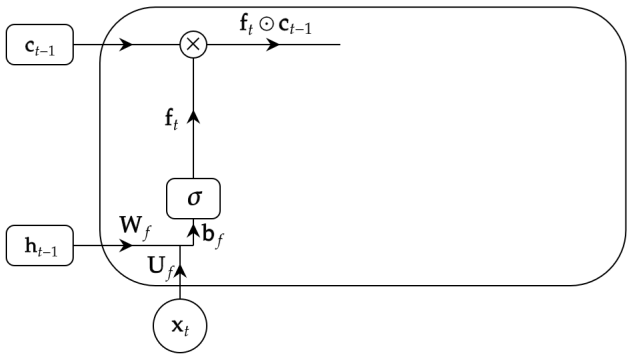
We can interpret the vector $\mathbf{h}_{t-1}$ as the **short term memory**, or **hidden state**, that contains information used to predict the next output value. However, this can be corrupted (e.g. forgetting information from many steps ago), so we add an additional **long term memory**, or **cell 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.

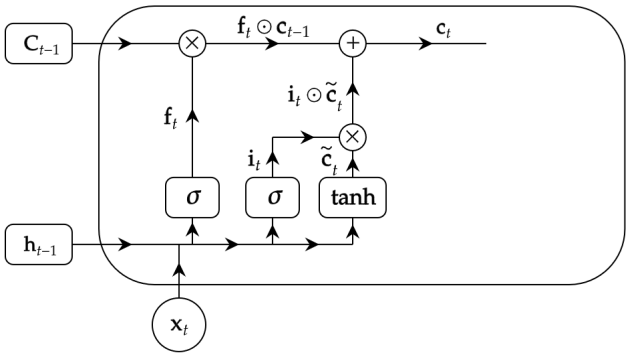
$$\mathbf{f}_t = \boldsymbol{\sigma}(\mathbf{W}_f \mathbf{h}_{t-1} + \mathbf{U}_f \mathbf{x}_t + \mathbf{b}_f)$$



2. The **input gate** (denoted by **i**) consists of two activations with the following operations.

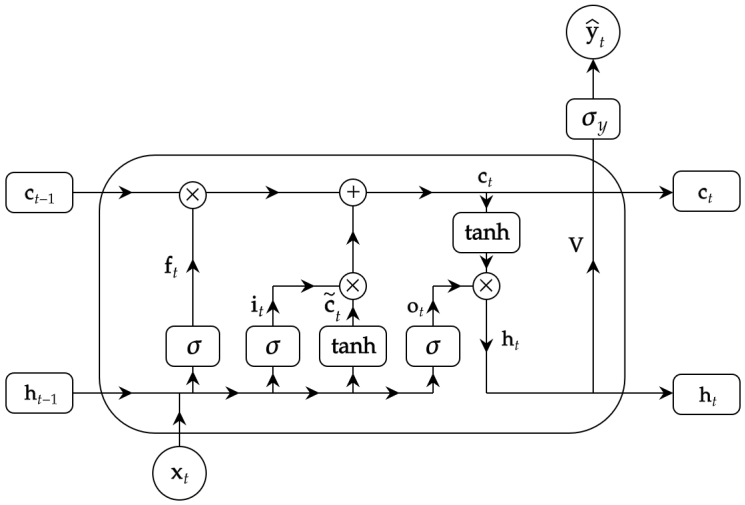
$$\mathbf{i}_t = \boldsymbol{\sigma}(\mathbf{W}_i \mathbf{h}_{t-1} + \mathbf{U}_i \mathbf{x}_t + \mathbf{b}_i)$$
$$\tilde{\mathbf{c}}_t = \mathbf{tanh}(\mathbf{W}_c \mathbf{h}_{t-1} + \mathbf{U}_c \mathbf{x}_t + \mathbf{b}_c)$$
$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot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

The layer **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 **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.

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



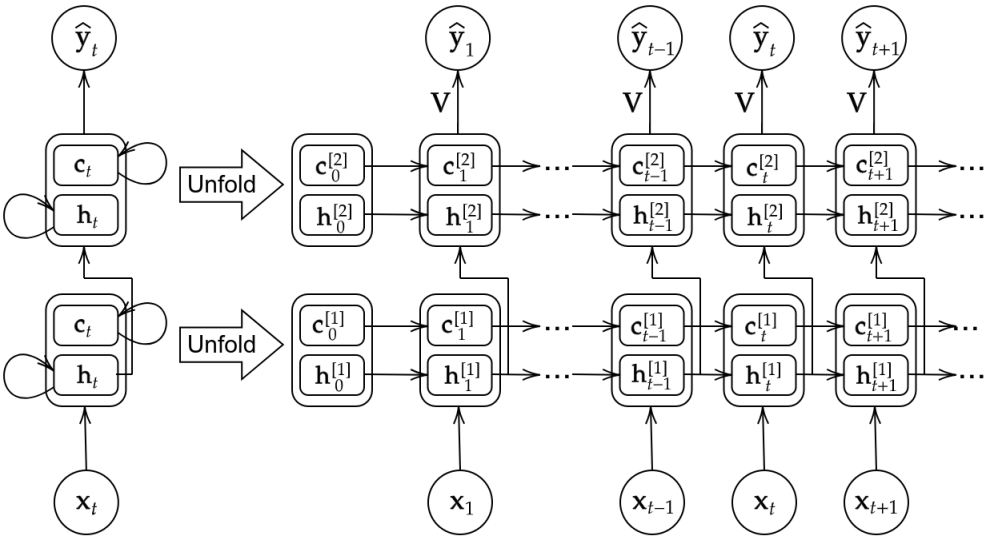
That is it! Now focusing on the cell state in the diagram above. Note that in order to go from cell state $\mathbf{c}_{t-1}$ to $\mathbf{c}_t$, there was not a whole lot done to it. We really just multiply it once, which potentially deletes some content, and add it once, which adds new content, and we 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.

5.3.1 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.

5.4 Gated Recurrent Units

6 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:

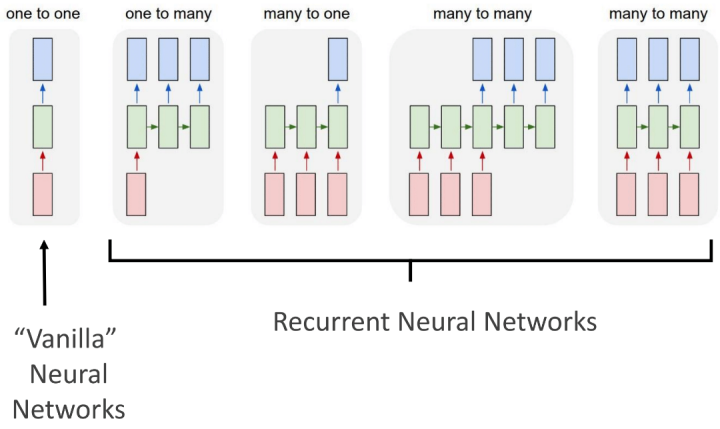
1. The input vector may not be in the correct form that we want. This is the motivation for the *seq2seq* model, where we are working with sequences of vectors that suffer from the problem of *locality* in RNNs. Therefore, it is necessary to encode this entire sequence into one vector, at the loss of 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.

6.1 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 $\mathbf{C} = \mathbf{h}_T$.

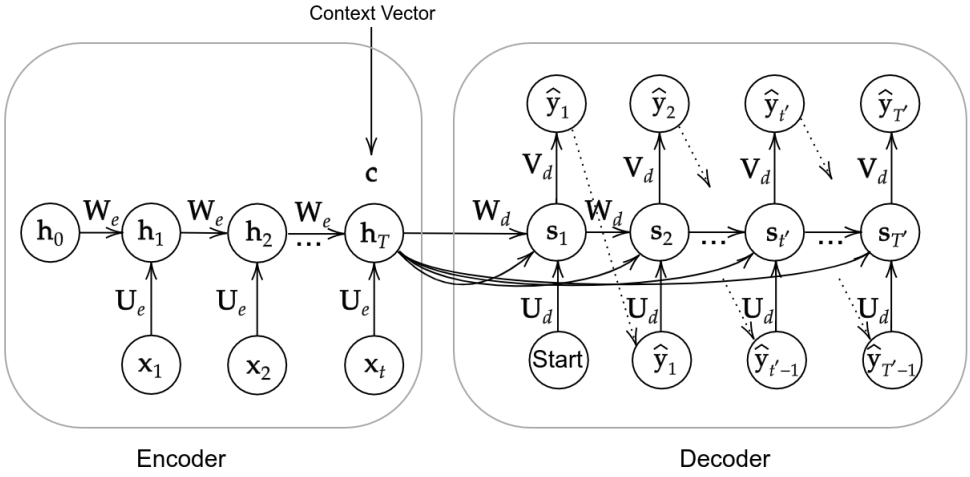
2. Now, given the target output sequence $\hat{\mathbf{y}}_1, \dots, \hat{\mathbf{y}}_{t'+1}$ for each timestep t' (we use t' to differentiate from the input sequence time steps), the decoder assigns a predicted probability to each possible token occurring at step $\hat{\mathbf{y}}_{t'+1}$ conditioned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

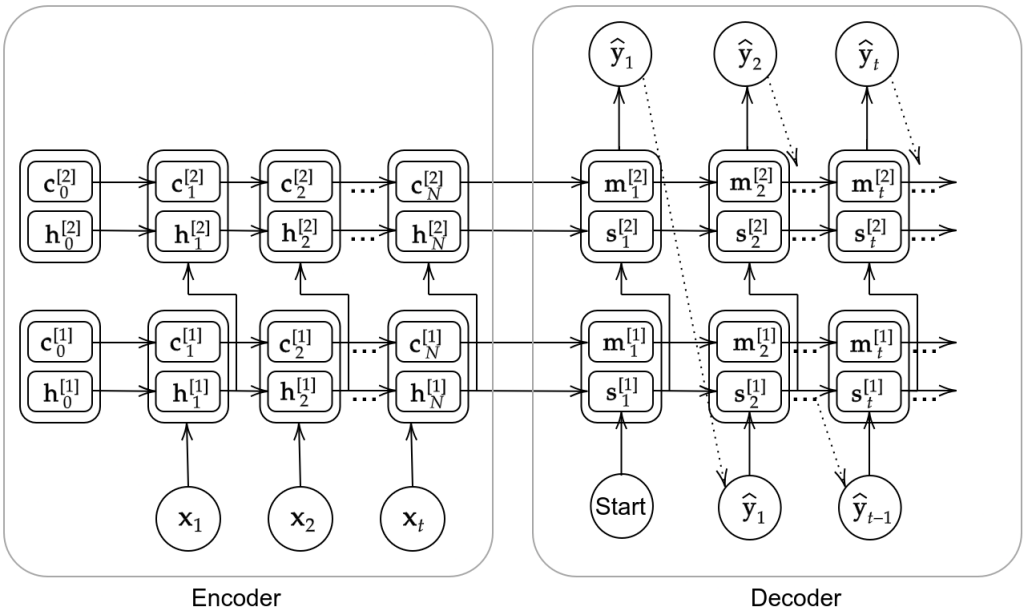
$$\mathbf{s}_{t'+1} = g(\mathbf{s}_{t'}, \hat{\mathbf{y}}_{t'}, \mathbf{C})$$

   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.

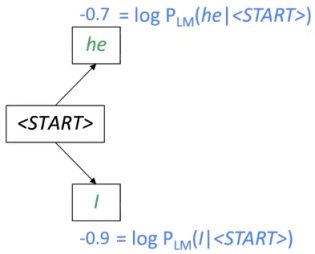
6.1.1 Decoding Schemes

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

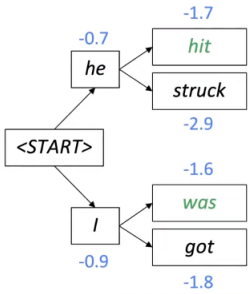
$$\mathbb{P}(y \mid x) = \prod_{t=1}^{T} \mathbb{P}(y_t \mid y_1, \dots, y_{t-1}, x)$$

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.

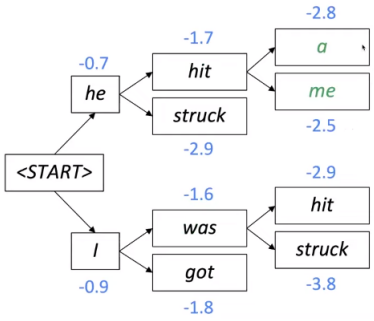
1. Given the START token, say that the k most probable next words were "he" (-0.7) and "I" (-0.9).



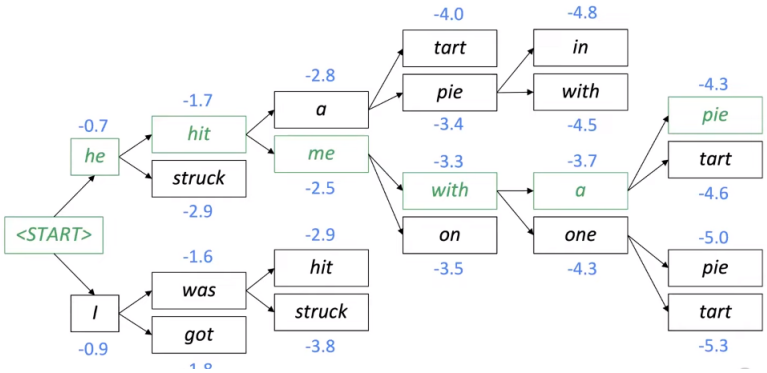
2. Now we look at the two most likely next words for each of "he" and "I" and out of the four possibilities, we compute the two most likely ones, which is "he hit" (-1.7) and "I was" (-1.6).



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.



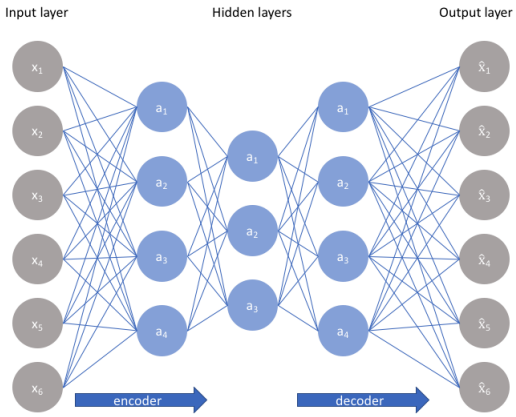
One more condition to mention is when to stop generating words. In greedy decoding, we usually decode until the model produces an END token. In beam search decoding, different hypotheses may produce END tokens on different timestamps, and so every time we have a complete hypothesis, we can place it aside 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).

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

6.2 Autoencoders

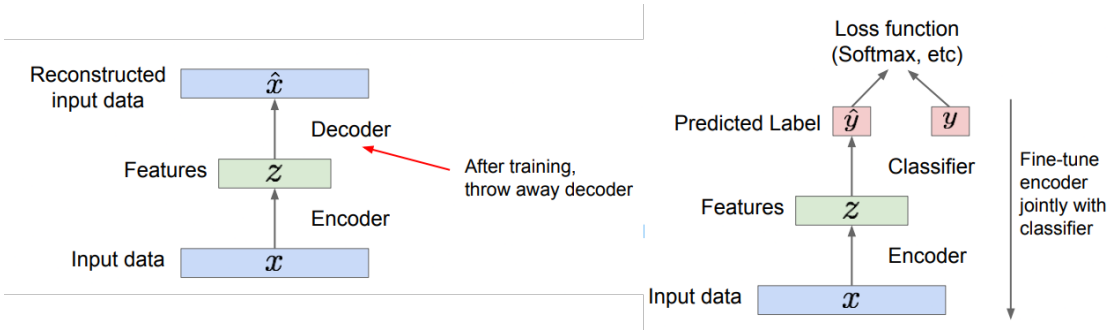
We can start with MLPs for autoencoders, as they naturally extend to other neural nets. As the name suggests, autoencoders literally means encoding itself. What we want to do is use a MLP encoder $\mathbf{f}$ to map our input $\mathbf{x}$ into a lower dimensional vector representation $\mathbf{z}$, which represents the meaningful features of our input. Now, we want to train our decoder $\mathbf{g}$ so that the features can be used to reconstruct the original data, which essentially performs some dimensionality reduction on the original input to produce $\hat{\mathbf{x}}$. To train this, we want to look at the difference between $\mathbf{x}$ and $\hat{\mathbf{x}}$, so we can train on, say the L2 loss function

$$L(\boldsymbol{\theta}) = ||\mathbf{x} - \hat{\mathbf{x}}||^2 = ||\mathbf{x} - \mathbf{g}_{\boldsymbol{\theta}}(\mathbf{f}_{\boldsymbol{\theta}}(\mathbf{x}))||^2$$



When autoencoders were first developed, researchers have originally worked with setting $\mathbf{f}$ and $\mathbf{g}$ as a simple linear plus activation function, but later on, the autoencoder architecture transitioned to using deep, fully-connected neural networks and CNNs.

Once we have trained our autoencoder, this becomes useful in transfer learning. After training the entire double network, we can throw away the decoder model. The encoder itself is now optimized to extract all relevant features, so we can simply attach on a classification layer to the end of the encoder network to initialize a supervised model. Then we can fine tune the entire model jointly with the classifier for effective training.



6.3 Image Captioning

In image captioning, we want our model to look at an image and generate some text relating to that image. This sounds like we can use an encoder CNN to extract features from the data and then use a decoder RNN to generate text based on this latent vector!


"man in black shirt is playing guitar."


"construction worker in orange safety vest is working on road."


"two young girls are playing with lego toy."

For higher performance gains, we use an *attention mechanism* to focus on different parts of the image while generating each word of the caption.

7 Attention Models

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).

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.

Given a set of input vector **values** and a vector **query**, attention is a technique to compute a weighted sum of the values, dependent on the query. 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).

1. The query is looking for information somewhere
2. The key is interacting with the query
3. The value is the thing that you're actually going to weigh in your average and output, i.e. the embedding vector.

7.1 Seq2Seq with Attention

A huge issue with the sequence-to-sequence model is the **bottleneck problem**. The encoder encodes the input sentence into a single latent vector at the end, and this one vector needs to capture *all information* about the source sentence. Clearly, if this vector is not large enough, we have too little bandwidth to capture this information, resulting in an information bottleneck.

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. Let's assume we have some seq2seq model with encoder hidden states $\mathbf{h}_1, \dots, \mathbf{h}_T \in \mathbb{R}^h$. On decoder timestep t', we have the decoder hidden state $\mathbf{s}_{t'} \in \mathbb{R}^s$, where s does not necessarily equal h since they can be completely different RNNs. We then compute the **attention scores** that determine which encoder hidden state we should pay attention to.

$$\mathbf{e}^{t'} = \left[\text{score}(\mathbf{s}_{t'}, \mathbf{h}_1), \dots, \text{score}(\mathbf{s}_{t'}, \mathbf{h}_T)\right] \in \mathbb{R}^T$$

where the simplest case is when $s = h$ and we can simply compute the dot product

$$\text{score}(\mathbf{s}_{t'}, \mathbf{h}_t) = \mathbf{s}_{t'} \cdot \mathbf{h}_t$$

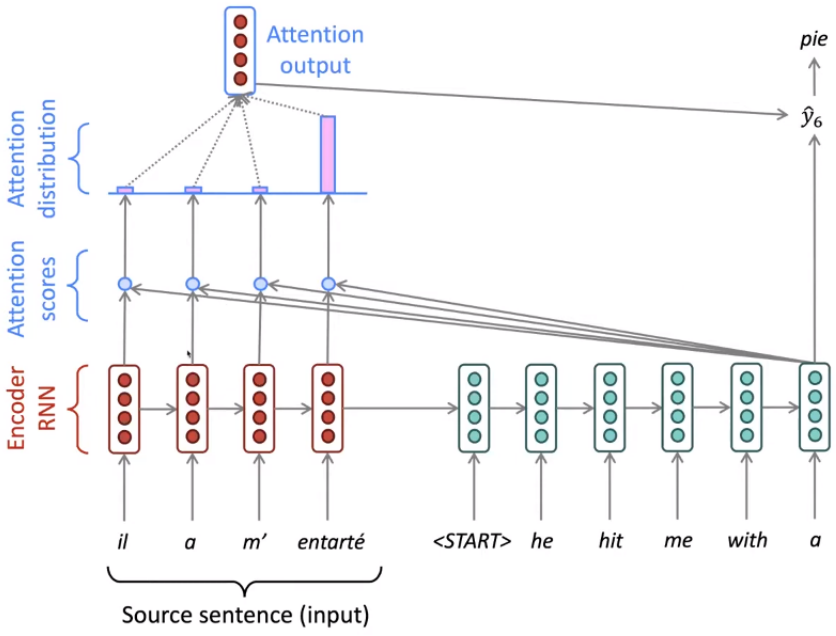
We take its softmax to get the **attention distribution** $\boldsymbol{\alpha}^{t'}$ for this step (a discrete probability distribution)

$$\boldsymbol{\alpha}^{t'} = \text{softmax}(\mathbf{e}^{t'}) \in \mathbb{R}^T$$

We use $\boldsymbol{\alpha}^{t'}$ to take a weighted sum of the encoder hidden states to get the attention output $\mathbf{a}_t$

$$\mathbf{a}_{t'} = \sum_{t=1}^{T} \alpha_t^{t'} \mathbf{h}_t \in \mathbb{R}^h$$

which acts as our context vector $\mathbf{C}_{t'}$ that we can now use in our vanilla seq2seq model. 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.



There are many score functions that we can choose from, with some listed:

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 $k << s, h$.

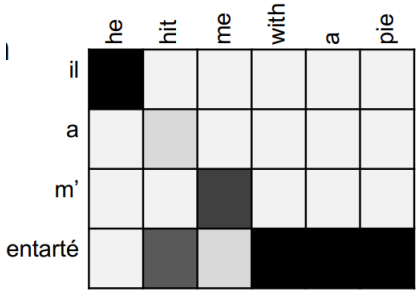
   $$e_t^{t'} = \text{score}(\mathbf{s}_{t'}, \mathbf{h}_t) = \mathbf{s}_{t'}^T (\mathbf{U}_a^T \mathbf{V}_a) \mathbf{h}_t = (\mathbf{U}_a \mathbf{s}_{t'})^T (\mathbf{V}_a \mathbf{h}_t) \in \mathbb{R}$$

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.

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).



7.2 Self-Attention

At first glance, this just sounds like a fully connected layer, but the main difference between the two is that in an attention model, the weights are dynamic w.r.t. the actually vector. Remember that in a fully connected layer, we have a bunch of weights that we're slowly learning over the training course. In attention, it's the actual interactions between the key and the query vectors which are dependent on the actual content, that are allowed to vary by time. And so the actual strengths of all these interactions, i.e. the attention weights (which are analogous to the weights of the fully connected layer), are allowed to change as a function of the input. So the attention weights are allowed to change as a function of the input. A separate difference is that the parameterizations are completely different.

8 Transformers

9 Generative Models

9.1 Variational Autoencoders

9.2 Generative Adversarial Networks (GANs)

10 Deep Reinforcement Learning