Multilayer Perceptrons

Muchang Bahng

June 23, 2023

Contents

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

1 Multi-Layered Perceptron

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.

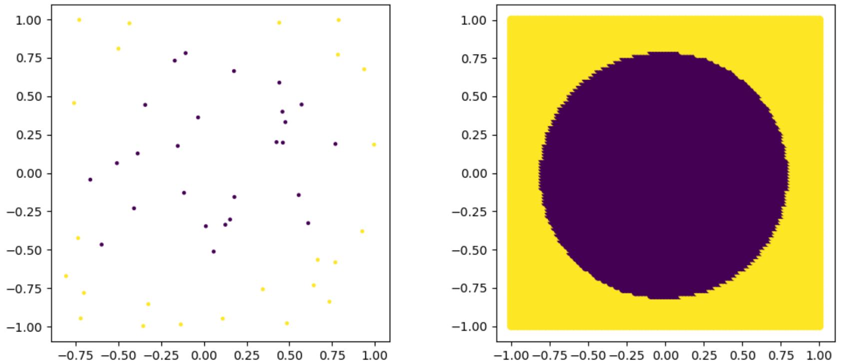
Definition 1.1 (Neuron)**.** A **neuron** is a function (visualized as a node) that takes in inputs $\mathbf{x}$ and outputs a value y calculated

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

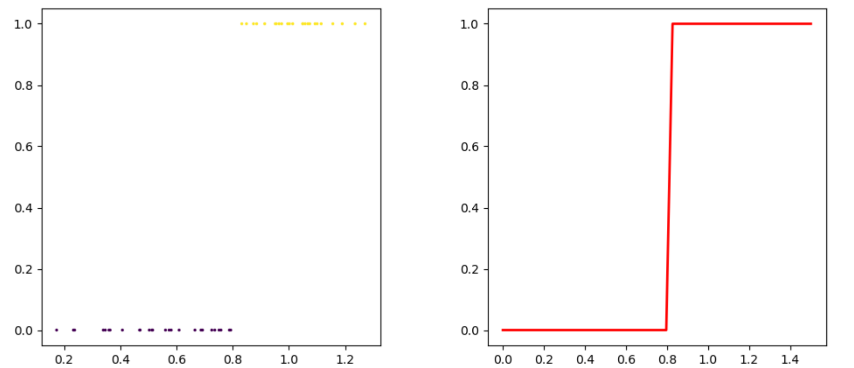
where σ is an activation function. Activation functions are usually simple functions with a range of $[0, 1]$ or $[-1, 1]$, and popular ones include:

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 $\mathbb{R}^{N^{[0]}} = \mathbb{R}^D$.

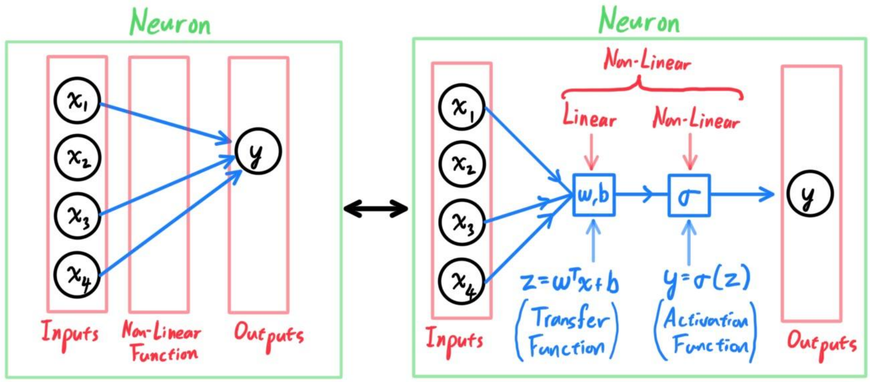
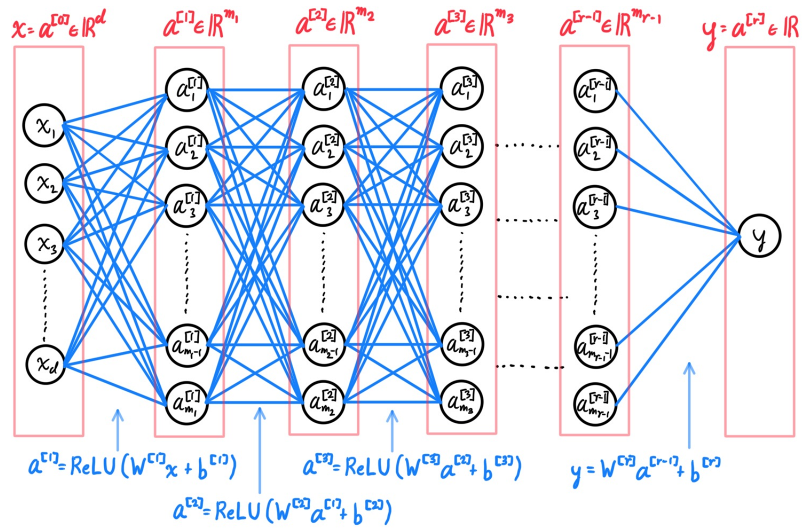


Figure 2: Diagram of a neuron, decomposed into its linear and nonlinear components.

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: $\mathbf{a}^{[L]} = \mathbf{g}(\mathbf{z}^{[L]}) = \mathbf{h}_{\boldsymbol{\theta}}(\mathbf{x}) \in \mathbb{R}^M$.

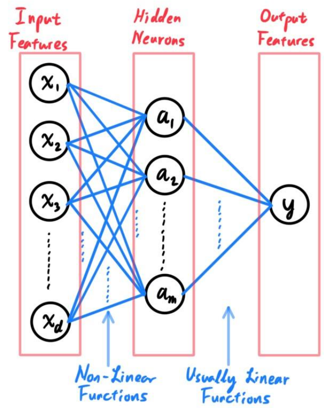
All the parameters of the neural net will be denoted $\boldsymbol{\theta}$. Ultimately, a neural net is really just a generalized linear model on an feature space with a ton of nonlinear preprocessing.



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 $a_1, \ldots, a_m$.



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 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 1.2 (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 is too positive or negative, the gradient will vanish, making very small updates.
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

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

   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 1.3 (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 1.4 (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 1.5 (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 1.6 (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 1.7 (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

1.2 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.

1.3 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.

1.4 Weight Space Symmetries

2 Network Training

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

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

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

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

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.

2.1 Backpropagation

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

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

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

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

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

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

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

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

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

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

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

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

2.1.1 Summary

Therefore, let us summarize what a MLP does:

1. *Initialization*: We initialize all the parameters to be
$$\boldsymbol{\theta} = (\mathbf{W}^{[1]}, \mathbf{b}^{[1]}, \mathbf{W}^{[2]}, \ldots, \mathbf{W}^{[L]}, \mathbf{b}^{[L]})$$
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
$$\mathbf{a}^{[0]} = \mathbf{x}^{(n)}, \mathbf{z}^{[1]}, \mathbf{a}^{[1]}, \ldots, \mathbf{z}^{[L]}, \mathbf{a}^{[L]} = h_{\boldsymbol{\theta}}(\mathbf{x}^{(n)})$$
4. *Back Propagation*:
   (a) Evaluate the $\boldsymbol{\delta}^{[l]}$'s starting from the back with the formula
$$\begin{aligned} \boldsymbol{\delta}^{[L]} &= \left(\frac{\partial \mathbf{g}}{\partial \mathbf{z}^{[L]}}\right)^T \left(\frac{\partial E_n}{\partial \mathbf{a}^{[L]}}\right) \\ \boldsymbol{\delta}^{[l]} &= \boldsymbol{\sigma}'(\mathbf{z}^{[l]}) \odot (\mathbf{W}^{[l+1]})^T \boldsymbol{\delta}^{[l+1]} \qquad l = 1, \ldots, L-1 \end{aligned}$$
   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 $l = 1, \ldots, L$,
$$\mathbf{W}^{[l]} = \mathbf{W}^{[l]} - \alpha \frac{\partial E_n}{\partial \mathbf{W}^{[l]}}, \qquad \mathbf{b}^{[l]} = \mathbf{b}^{[l]} - \alpha \frac{\partial E_n}{\partial \mathbf{b}^{[l]}}$$
   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.

2.2 Implementation of Neural Net in Python

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
```

2.3 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.

2.3.1 Cuda Support

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")
```

2.3.2 Datasets

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 $(0, 1, \ldots, 9)$. 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])
```

2.3.3 Feature and Label Transformations

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.

2.3.4 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")
```

2.3.5 Automatic Differentiation

2.3.6 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!")
```

3 Regularization and Stability

3.1 Early Stopping

3.2 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.

3.3 L1 and L2 Regularization

3.4 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.

3.5 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

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

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 $\mathcal{B} = \{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(B)}\}$. 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,

$$\mu_d = \frac{1}{B} \sum_b x_d^{(b)}, \;\; \sigma_d^2 = \frac{1}{B} \sum_b \big(x_d^{(b)} - \mu_d\big)^2$$

   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 $\gamma_d = \sqrt{\mathrm{Var}(x_d)}$ 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

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

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.

3.6 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.

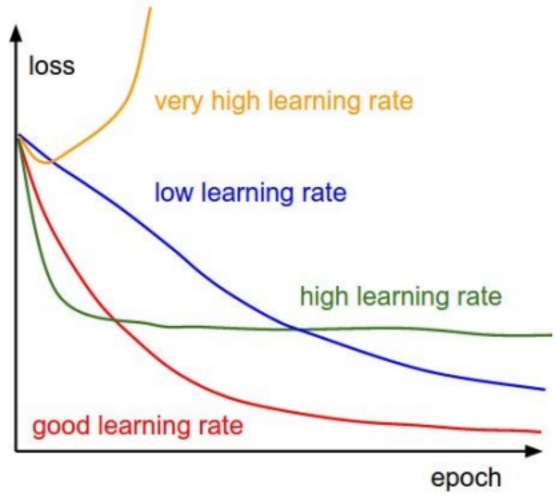
3.7 Sharpness Aware Minimization

https://openreview.net/pdf?id=6Tm1mposlrM

3.8 Babysitting the Learning Process

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 Network Pruning

5 Neural Additive Models

Generalized additive models.

https://r.qcbs.ca/workshop08/book-en/gam-with-interaction-terms.html

https://arxiv.org/pdf/2004.13912.pdf

6 Lipshitz Regularity of Deep Neural Networks

Deep neural networks are known for being overparameterized and tends to predict data very nicely, known as benign overfitting. In fact, it can be proved that a data set of any size, we can always fit a one-layer perceptron that perfectly fits through all of them, given that the layer is large enough. In most cases, we are interested in fitting the data *smoothly* in the sense that data extrapolations are stable, i.e. a small perturbation of x should result in a small perturbation of $h(x)$. It turns out that the more parameters it has, the better this stability is and therefore the more robust the model.

Deep neural networks, despite their usefulness in many problems, are known for being very sensitive to their input. Adversarial examples take advantage of this weakness by adding carefully chosen perturbations that drastically change the output of the network. Adversarial machine learning attempts to study these weaknesses and hopefully use them to create more robust models. It is natural to expect that the precise configuration of the minimal necessary perturbations is a random artifact of the normal variability that arises in different runs of backpropagation learning. Yet, it has been found that adversarial examples are relatively robust, and are shared by neural networks with varied number of layers, activations or trained on different subsets of the training data. This suggest that the deep neural networks that are learned by backpropagation have *intrinsic* blind spots, whose structure is connected to the data distribution in a non-obvious way.

A metric to assess the robustness of a deep neural net $h_\theta : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ is its Lipshitz constant, which effectively bounds how much h can change given some change in $\mathbf{x}$.

Definition 6.1 (Lipshitz Continuity)**.** A function $f : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ is called **Lipshitz continuous** if there exists a constant L such that for all $x, y \in \mathbb{R}^n$

$$||f(x) - f(y)||_2 \leq L||x - y||_2$$

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

Theorem 6.1. If $f : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ is Lipschitz continuous, then

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

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

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

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

Theorem 6.2 (Lipschitz Upper Bound for MLPs)**.** It has already been shown that for a K-layer MLP

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

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

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

$$\begin{aligned}
v_0(\mathbf{x}) &= \mathbf{x} \\
v_1(\mathbf{x}) &= g_1\big(v_0(\mathbf{x})\big) \\
v_2(\mathbf{x}) &= g_2\big(v_0(\mathbf{x}), v_1(\mathbf{x})\big) \\
\ldots &= \ldots \\
v_k(\mathbf{x}) &= g_k\big(v_0(\mathbf{x}), v_1(\mathbf{x}), \ldots, v_{k-1}(\mathbf{x})\big) \\
\ldots &= \ldots \\
v_K(\mathbf{x}) &= g_K\big(v_0(\mathbf{x}), v_1(\mathbf{x}), \ldots, v_{K-2}(\mathbf{x}), v_{K-1}(\mathbf{x})\big)
\end{aligned}$$

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

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

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

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

Now we can compute the maximum iteratively.

1. First,

$$\frac{\partial v_0}{\partial \mathbf{x}} = I_{n \times n} \implies \bigg|\bigg| \frac{\partial v_0}{\partial \mathbf{x}} \bigg|\bigg|_2 = 1$$

2. Second,
$$\Bigg|\Bigg| \underbrace{\frac{\partial v_1}{\partial \mathbf{x}}}_{n_1 \times n} \Bigg|\Bigg| \leq \Bigg|\Bigg| \underbrace{\frac{\partial g_1}{\partial v_0}}_{n_1 \times n} \Bigg|\Bigg| \Bigg|\Bigg| \underbrace{\frac{\partial v_0}{\partial \mathbf{x}}}_{n \times n} \Bigg|\Bigg| = \left|\left|\frac{\partial g_1}{\partial v_0}\right|\right| \cdot ||I|| = \left|\left|\frac{\partial g_1}{\partial v_0}\right|\right|$$

3. Third,
$$\begin{aligned} \Bigg|\Bigg| \underbrace{\frac{\partial v_2}{\partial \mathbf{x}}}_{n_2 \times n} \Bigg|\Bigg| &\leq \Bigg|\Bigg| \underbrace{\frac{\partial g_2}{\partial v_1}}_{n_2 \times n_1} \Bigg|\Bigg| \Bigg|\Bigg| \underbrace{\frac{\partial v_1}{\partial \mathbf{x}}}_{n_1 \times n} \Bigg|\Bigg| + \Bigg|\Bigg| \underbrace{\frac{\partial g_2}{\partial v_0}}_{n_2 \times n_0} \Bigg|\Bigg| \Bigg|\Bigg| \underbrace{\frac{\partial v_0}{\partial \mathbf{x}}}_{n_0 \times n} \Bigg|\Bigg| \\ &= \left|\left|\frac{\partial g_2}{\partial v_1}\right|\right| \left|\left|\frac{\partial g_1}{\partial v_0}\right|\right| + \left|\left|\frac{\partial g_2}{\partial v_0}\right|\right| 1 \end{aligned}$$

4. and so on, where we have calculated all previous $\frac{\partial v_i}{\partial \mathbf{x}}$ for $i \in [k-1]$, and we just need to compute $\frac{\partial g_k}{\partial v_i}$ for $i \in [k-1]$.