Muchang Bahng

Spring 2023

Contents

Here is a few steps you can take as a guide to training a neural network.[1]

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.

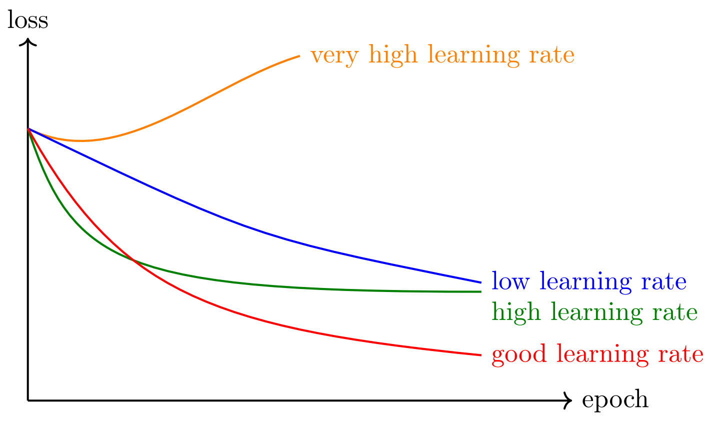


Figure 1: If you see loss curves that are flat for a while and then start decreasing, then bad initialization is a prime suspect.

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

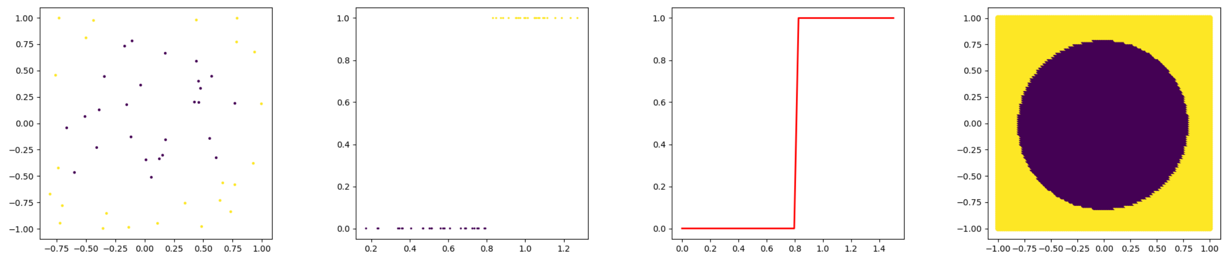
[1]From Stanford CS 229 NLP.

1 Multi-Layered Perceptrons

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

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

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



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

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

1.1 Feedforward Fully-Connected Networks

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

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

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

Definition 1.1 (Neuron)

A **neuron** is a function of form

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

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

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

Definition 1.2 (Feedforward, Fully-Connected Multilayer Perceptron)

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

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

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

1. The inputs will be labeled $\mathbf{x} = a^{[0]}$ which is in $\mathbb{R}^{N^{[0]}} = \mathbb{R}^D$.
2. We map $a^{[l]} \in \mathbb{R}^{N^{[l]}} \mapsto W^{[l+1]}a^{[l]} + b^{[l+1]} = z^{[l+1]} \in \mathbb{R}^{N^{[l+1]}}$, where z denotes a vector after an affine transformation.
3. We map $z^{[l+1]} \in \mathbb{R}^{N^{[l+1]}} \mapsto \sigma(z^{[l+1]}) = a^{[l+1]} \in \mathbb{R}^{N^{[l+1]}}$, where a denotes a vector after an activation function.
4. We keep doing this until we reach the second last layer with vector $a^{[L-1]}$. Note that in the last layer we do *not* apply an activation function.

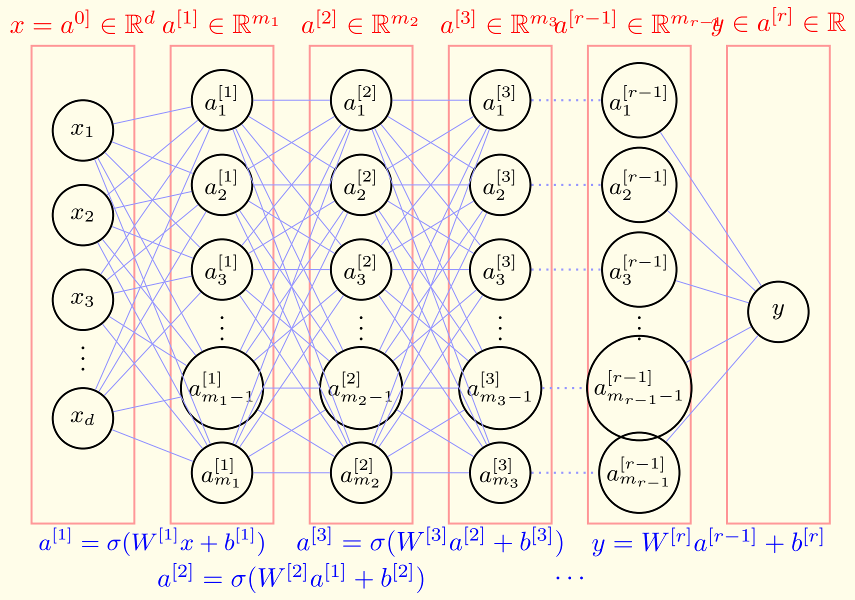

$x = a^{0]} \in \mathbb{R}^d$
$a^{[1]} \in \mathbb{R}^{m_1}$
$a^{[2]} \in \mathbb{R}^{m_2}$
$a^{[3]} \in \mathbb{R}^{m_3}$
$a^{[r-1]} \in \mathbb{R}^{m_{r-1}}$
$y \in a^{[r]} \in \mathbb{R}$
x_1
x_2
x_3
x_d
$a_1^{[1]}$
$a_2^{[1]}$
$a_3^{[1]}$
$a_{m_1-1}^{[1]}$
$a_{m_1}^{[1]}$
$a_1^{[2]}$
$a_2^{[2]}$
$a_3^{[2]}$
$a_{m_2-1}^{[2]}$
$a_{m_2}^{[2]}$
$a_1^{[3]}$
$a_2^{[3]}$
$a_3^{[3]}$
$a_{m_3-1}^{[3]}$
$a_{m_3}^{[3]}$
$a_1^{[r-1]}$
$a_2^{[r-1]}$
$a_3^{[r-1]}$
$a_{m_{r-1}-1}^{[r-1]}$
$a_{m_{r-1}}^{[r-1]}$
y
$a^{[1]} = \sigma(W^{[1]}x + b^{[1]})$
$a^{[2]} = \sigma(W^{[2]}a^{[1]} + b^{[2]})$
$a^{[3]} = \sigma(W^{[3]}a^{[2]} + b^{[3]})$
$\cdots$
$y = W^{[r]}a^{[r-1]} + b^{[r]}$

Figure 3: If there does not exist any edge from a potential input x to an output y, then this means that x is not relevant in calculating y, i.e. the weight is 0. 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.

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

Code 1.1 (Parameters and Neural Nets in PyTorch)

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

1.2 Forward and Back Propagation

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

Definition 1.3 (Forward Propagation)

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

$$x = a^{[0]} \mapsto z^{[1]} \mapsto a^{[1]} \mapsto \ldots \mapsto z^{[L]} \mapsto a^{[L]} = f(x) \tag{5}$$

is called **forward propagation**.

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

$$\frac{\partial f}{\partial z^{[l]}} = \frac{\partial f}{\partial z^{[l+1]}} \cdot \frac{z^{[l+1]}}{z^{[l]}} \tag{6}$$

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

Definition 1.4 (Backward Propagation)

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

$$\frac{\partial f}{\partial a^{[L]}} \mapsto \frac{\partial f}{\partial z^{[L]}} \mapsto \ldots \mapsto \frac{\partial f}{\partial z^{[L]}}, \tag{7}$$

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

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

Algorithm 1.1 (Backpropagation)

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

$$\frac{\partial E_n}{\partial w_{ji}^{[l]}} = \underbrace{\frac{\partial E_n}{\partial \mathbf{z}^{[l]}}}_{1\times N^{[l]}} \cdot \underbrace{\frac{\partial \mathbf{z}^{[l]}}{\partial w_{ji}^{[l]}}}_{N^{[l]}\times 1} \text{ and } \frac{\partial E_n}{\partial b_i^{[l]}} = \underbrace{\frac{\partial E_n}{\partial \mathbf{z}^{[l]}}}_{1\times N^{[l]}} \cdot \underbrace{\frac{\partial \mathbf{z}^{[l]}}{\partial b_i^{[l]}}}_{N^{[l]}\times 1} \tag{8}$$

It helps to visualize that we are focusing on

$$\mathbf{h}_{\boldsymbol{\theta}}(\mathbf{x}) = g\big(\ldots \sigma(\underbrace{\mathbf{W}^{[l]}\mathbf{a}^{[l-1]} + \mathbf{b}^{[l]}}_{\mathbf{z}^{[l]}}) \ldots\big) \tag{9}$$

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

$$\mathbf{z}^{[l]} = \begin{pmatrix} w_{11}^{[l]} & \dots & w_{1N^{[l-1]}}^{[l]} \\ \vdots & \ddots & \vdots \\ w_{N^{[l]}1}^{[l]} & \dots & w_{N^{[l]}N^{[l-1]}}^{[l]} \end{pmatrix} \begin{pmatrix} a_1^{[l-1]} \\ \vdots \\ a_{N^{[l-1]}}^{[l-1]} \end{pmatrix} + \begin{pmatrix} b_1^{[l]} \\ \vdots \\ b_{N_{[l]}^{[l]}} \end{pmatrix} \tag{10}$$

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

$$\frac{\partial E_n}{\partial w_{ji}^{[l]}} = \underbrace{\frac{\partial E_n}{\partial z_j^{[l]}}}_{1\times 1} \cdot \underbrace{\frac{\partial z_j^{[l]}}{\partial w_{ji}^{[l]}}}_{1\times 1} = \delta_j^{[l]} \cdot \frac{\partial z_j^{[l]}}{\partial w_{ji}^{[l]}} \text{ and } \frac{\partial E_n}{\partial b_i^{[l]}} = \underbrace{\frac{\partial E_n}{\partial z_j^{[l]}}}_{1\times 1} \cdot \underbrace{\frac{\partial z_j^{[l]}}{\partial b_i^{[l]}}}_{1\times 1} = \delta_j^{[l]} \cdot \frac{\partial z_j^{[l]}}{\partial b_i^{[l]}} \tag{11}$$

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

$$z_j^{[l]} = w_{j1}^{[l]} a_1^{[l-1]} + \ldots w_{jN^{[l-1]}} a_{N^{[l-1]}}^{[l-1]} + b_j^{[l]} \tag{12}$$

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

$$\frac{\partial E_n}{\partial w_{ji}^{[l]}} = \delta_j^{[l]} a_i^{[l-1]}, \qquad \frac{\partial E_n}{\partial b_i^{[l]}} = \delta_j^{[l]} \tag{13}$$

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

$$\delta_j^{[L]} = \frac{\partial E_n}{\partial z_j^{[L]}} = \frac{\partial E_n}{\partial \mathbf{a}^{[L]}} \cdot \frac{\partial \mathbf{a}^{[L]}}{\partial z_j^{[L]}} = \sum_k \frac{\partial E_n}{\partial a_k^{[L]}} \cdot \frac{\partial a_k^{[L]}}{\partial z_j^{[L]}} \tag{14}$$

which can be rewritten in the matrix notation

$$\boldsymbol{\delta}^{[L]} = \left(\frac{\partial \mathbf{g}}{\partial \mathbf{z}^{[L]}}\right)^T \left(\frac{\partial E_n}{\partial \mathbf{a}^{[L]}}\right) = \underbrace{\begin{bmatrix} \frac{\partial g_1}{\partial z_1^{[L]}} & \dots & \frac{\partial g_{N^{[L]}}}{\partial z_1^{[L]}} \\ \vdots & \ddots & \vdots \\ \frac{\partial g_1}{\partial z_{N^{[L]}}^{[L]}} & \dots & \frac{\partial g_{N^{[L]}}}{\partial z_{N^{[L]}}^{[L]}} \end{bmatrix}}_{N^{[L]}\times N^{[L]}} \begin{bmatrix} \frac{\partial E_n}{\partial a_1^{[L]}} \\ \vdots \\ \frac{\partial E_n}{\partial a_{N^{[L]}}^{[L]}} \end{bmatrix} \tag{15}$$

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

1. If we assume that $Y \mid X = x$ is Gaussian in a regression (of scalar output) setting, then our canonical link would be $g(x) = x$, which gives the sum of squares loss function. Note that since the output is a real-valued scalar, $\mathbf{a}^{[L]}$ will be a scalar (i.e. the final layer is one node, $N^{[L]} = 1$).
$$E_n = \frac{1}{2}(y^{(n)} - a^{[L]})^2 \tag{16}$$
To calculate $\boldsymbol{\delta}^{[L]}$, we can simply use the chain rule to get
$$\delta^{[L]} = \frac{\partial E_n}{\partial z^{[L]}} = \frac{\partial E_n}{\partial a^{[L]}} \cdot \frac{\partial a^{[L]}}{\partial z^{[L]}} = a^{[L]} - y^{(n)} \tag{17}$$
2. For classification (of M classes), we would use the softmax activation function (with its derivative next to it for convenience)
$$\mathbf{g}(\mathbf{z}) = \mathbf{g}\left(\begin{bmatrix} z_1 \\ \vdots \\ z_M \end{bmatrix} \right) = \begin{bmatrix} e^{z_1} / \sum_k e^{z_k} \\ \vdots \\ e^{z_M} / \sum_k e^{z_k} \end{bmatrix}, \quad \frac{\partial g_k}{\partial z_j} = \begin{cases} g_j(1 - g_j) & \text{if } k = j \\ -g_j g_k & \text{if } k \neq j \end{cases} \tag{18}$$
which gives the cross entropy error
$$E_n = -\mathbf{y}^{(n)} \cdot \ln\big(\mathbf{h}_{\boldsymbol{\theta}}(\mathbf{x}^{(n)})\big) = -\sum_i y_i^{(n)} \ln(a_i^{[L]}) \tag{19}$$
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
$$\delta_j^{[L]} = \sum_k \frac{\partial E_n}{\partial a_k^{[L]}} \cdot \frac{\partial a_k^{[L]}}{\partial z_j^{[L]}} \tag{20}$$
$$= -\sum_k \frac{y_k^{(n)}}{a_k^{[L]}} \cdot \frac{\partial a_k^{[L]}}{\partial z_j^{[L]}} \tag{21}$$
$$= \bigg(-\sum_{k \neq j} \frac{y_k^{(n)}}{a_k^{[L]}} \cdot \frac{\partial a_k^{[L]}}{\partial z_j^{[L]}} \bigg) - \frac{y_j^{(n)}}{a_j^{[L]}} \cdot \frac{a_j^{[L]}}{\partial z_j^{[L]}} \tag{22}$$
$$= \bigg(-\sum_{k \neq j} \frac{y_k^{(n)}}{a_k^{[L]}} \cdot -a_k^{[L]} a_j^{[L]} \bigg) - \frac{y_j^{(n)}}{a_j^{[L]}} \cdot a_j^{[L]}(1 - a_j^{[L]}) \tag{23}$$
$$= a_j^{[L]} \underbrace{\sum_k y_k^{(n)}}_{1} - y_j^{(n)} = a_j^{[L]} - y_j^{(n)} \tag{24}$$
giving us
$$\boldsymbol{\delta}^{[L]} = \mathbf{a}_j^{[L]} - \mathbf{y}^{[L]} \tag{25}$$

Now that we have found the error for the last layer, we can continue for the hidden layers. We can again expand by chain rule that
$$\delta_j^{[l]} = \frac{\partial E_n}{\partial z_j^{[l]}} = \frac{\partial E_n}{\partial \mathbf{z}^{[l+1]}} \cdot \frac{\partial \mathbf{z}^{[l+1]}}{\partial z_j^{[l]}} = \sum_{k=1}^{N^{[l+1]}} \frac{\partial E_n}{\partial z_k^{[l+1]}} \cdot \frac{\partial z_k^{[l+1]}}{\partial z_j^{[l]}} = \sum_{k=1}^{N^{[l+1]}} \delta_k^{[l+1]} \cdot \frac{\partial z_k^{[l+1]}}{\partial z_j^{[l]}} \tag{26}$$
By going backwards from the last layer, we should already have the values of $\delta_k^{[l+1]}$, and to compute the second partial, we recall the way a was calculated
$$z_k^{[l+1]} = b_k^{[l+1]} + \sum_{j=1}^{N^{[l]}} w_{kj}^{[l+1]} \sigma(z_j^{[l]}) \implies \frac{\partial z_k^{[l+1]}}{\partial z_j^{[l]}} = w_{kj}^{[l+1]} \cdot \sigma'(z_j^{[l]}) \tag{27}$$

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

$$\delta_j^{[l]} = \sigma'(z_j^{[l]}) \sum_{k=1}^{N^{[l+1]}} \delta_k^{[l+1]} \cdot w_{kj}^{[l+1]} \tag{28}$$

which gives the matrix form

$$\boldsymbol{\delta}^{[l]} = \boldsymbol{\sigma}'(\mathbf{z}^{[l]}) \odot (\mathbf{W}^{[l+1]})^T \boldsymbol{\delta}^{[l+1]} = \begin{bmatrix} \sigma'(z_1^{[l]}) \\ \vdots \\ \sigma'(z_{N^{[L]}}^{[l]}) \end{bmatrix} \odot \begin{bmatrix} w_{11}^{[l+1]} & \dots & w_{N^{[l+1]}1}^{[l+1]} \\ \vdots & \ddots & \vdots \\ w_{1N^{[l]}}^{[l+1]} & \dots & w_{N^{[l+1]}N^{[l]}}^{[l+1]} \end{bmatrix} \begin{bmatrix} \delta_1^{[l+1]} \\ \vdots \\ \delta_{N^{[l+1]}}^{[l+1]} \end{bmatrix} \tag{29}$$

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

$$\frac{\partial E_n}{\partial w_{ji}^{[l]}} = a_i^{[l-1]} \sigma'(z_j^{[l]}) \sum_{k=1}^{N^{[l+1]}} \delta_k^{[l+1]} \cdot w_{kj}^{[l+1]} \tag{30}$$

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

Algorithm 1.2 (Epoch of Training)

Before training, we initialize all the parameters to be

$$\boldsymbol{\theta} = (\mathbf{W}^{[1]}, \mathbf{b}^{[1]}, \mathbf{W}^{[2]}, \ldots, \mathbf{W}^{[L]}, \mathbf{b}^{[L]}) \tag{31}$$

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

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

$$\mathbf{a}^{[0]} = \mathbf{x}^{(n)}, \mathbf{z}^{[1]}, \mathbf{a}^{[1]}, \ldots, \mathbf{z}^{[L]}, \mathbf{a}^{[L]} = h_{\boldsymbol{\theta}}(\mathbf{x}^{(n)}) \tag{32}$$

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

$$\boldsymbol{\delta}^{[L]} = \left(\frac{\partial \mathbf{g}}{\partial \mathbf{z}^{[L]}}\right)^T \left(\frac{\partial E_n}{\partial \mathbf{a}^{[L]}}\right) \tag{33}$$

$$\boldsymbol{\delta}^{[l]} = \boldsymbol{\sigma}'(\mathbf{z}^{[l]}) \odot (\mathbf{W}^{[l+1]})^T \boldsymbol{\delta}^{[l+1]} \qquad l = 1, \ldots, L-1 \tag{34}$$

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

   (b) Calculate the derivatives of the error as

$$\frac{\partial E_n}{\partial \mathbf{W}^{[l]}} = \boldsymbol{\delta}^{[l]} (\mathbf{a}^{[l-1]})^T, \qquad \frac{\partial E_n}{\partial \mathbf{b}^{[l]}} = \boldsymbol{\delta}^{[l]} \tag{35}$$

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

$$\mathbf{W}^{[l]} = \mathbf{W}^{[l]} - \alpha \frac{\partial E_n}{\partial \mathbf{W}^{[l]}}, \qquad \mathbf{b}^{[l]} = \mathbf{b}^{[l]} - \alpha \frac{\partial E_n}{\partial \mathbf{b}^{[l]}} \tag{36}$$

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

Code 1.2 (Neural Net from Scratch)

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

Code 1.3 (Pytorch Implementation of Forward and Backward Propagation)

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

2 Theoretical Properties

2.1 Universal Approximation

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

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

Example 2.1 (Bump Functions in $\mathbb{R}$)

Assuming the sigmoid activation function is used, the bump function

$$f(x) = \begin{cases} 1 & \text{if } a < x < b \\ 0 & \text{if else} \end{cases} \tag{37}$$

can be approximated by taking a linear combination of a sigmoid function stepping up and one stepping down. That is,

$$f(x) \approx \frac{1}{2}\sigma\big(k(x-a)\big) - \frac{1}{2}\sigma\big(k(x-b)\big) \tag{38}$$

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

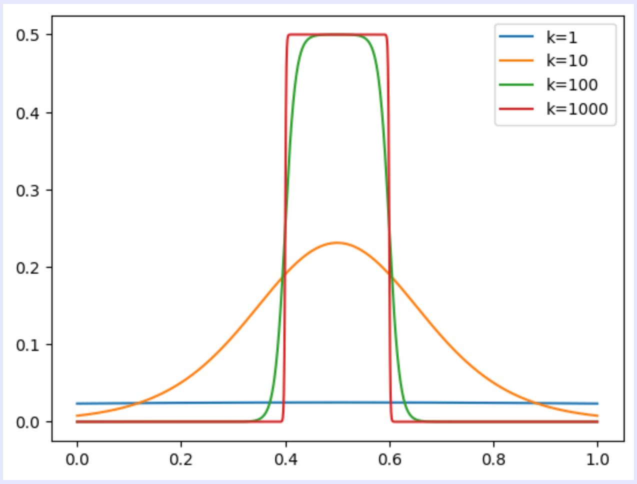


Figure 4: Bump function approximated with $a = 0.4, b = 0.6$, with differing values of k.

Example 2.2 (Bump Functions in $\mathbb{R}^2$)

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

$$f(x_1, x_2) = \begin{cases} 1 \text{ if } a < x_1 < b & \\ 0 & \text{if else} \end{cases} \tag{39}$$

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

$$f(x) \approx \frac{1}{2}\sigma\big(k(x_1 - a)\big) - \frac{1}{2}\sigma\big(k(x_1 - b)\big) \tag{40}$$

Example 2.3 (Tower Functions in $\mathbb{R}^2$)

Now to construct a tower function of the form

$$f(x_1, x_2) = \begin{cases} 1 & \text{if } a_1 < x_1 < b_1, a_2 < x_2 < b_2 \\ 0 & \text{if else} \end{cases} \tag{41}$$

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

$$f(x) \approx \sigma\Bigg(k_2\big[\sigma\big(k_1(x_1 - a_1)\big) - \sigma\big(k_1(x_1 - b_1)\big) + \sigma\big(k_1(x_2 - a_2)\big) - \sigma\big(k_1(x_2 - b_2)\big)big] - b_2\Bigg) \tag{42}$$

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

Theorem 2.1 (CS671 Fall 2023 PS5)

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

$$\sup_{x \in [0,1]^2} |f(x) - \tilde{f}(x)| \leq \epsilon \tag{43}$$

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

2.2 Parameter Symmetry

Early in the development of the theory of neural nets, An Mei Chen, (currently VP of engineering in Qualcomm) showed in [CLHN93] that for certain neural networks, there are multiple parameters θ that map to the same function f.

Theorem 2.2 (Parameter Symmetry)

Consider a 2-layer feedforward network of form

$$f = W^{[2]} \circ \sigma \circ W^{[1]} \tag{44}$$

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

2.3 Smoothness

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

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

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

Definition 2.1 (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 $x, y \in \mathbb{R}^n$

$$||f(x) - f(y)||_2 \leq L||x - y||_2 \tag{45}$$

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

Theorem 2.3 (Lipschitz Upper Bound with Operator Norm of Total Derivative)

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

$$\text{Lip}(f) = \sup_{x \in \mathbb{R}^n} ||D_x f||_{\text{op}} \tag{46}$$

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

$$\text{Lip}(f) = \sup_{x \in \mathbb{R}^n} ||\nabla f(x)||_2 \tag{47}$$

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 2.4 (Lipschitz Upper Bound for MLPs)

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

$$h_\theta(\mathbf{x}) \coloneqq \mathbf{T}_K \circ \boldsymbol{\rho}_{K-1} \circ \mathbf{T}_{K-1} \circ \cdots \circ \boldsymbol{\rho}_1 \circ \mathbf{T}_1(\mathbf{x}) \tag{48}$$

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.

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

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

$$v_0(\mathbf{x}) = \mathbf{x} \tag{49}$$
$$v_1(\mathbf{x}) = g_1\big(v_0(\mathbf{x})\big) \tag{50}$$
$$v_2(\mathbf{x}) = g_2\big(v_0(\mathbf{x}), v_1(\mathbf{x})\big) \tag{51}$$
$$\ldots = \ldots \tag{52}$$
$$v_k(\mathbf{x}) = g_k\big(v_0(\mathbf{x}), v_1(\mathbf{x}), \ldots, v_{k-1}(\mathbf{x})\big) \tag{53}$$
$$\ldots = \ldots \tag{54}$$
$$v_K(\mathbf{x}) = g_K\big(v_0(\mathbf{x}), v_1(\mathbf{x}), \ldots, v_{K-2}(\mathbf{x}), v_{K-1}(\mathbf{x})\big) \tag{55}$$

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

$$g_k : \prod_{i=0}^{k-1} \mathbb{R}^{n_i} \longrightarrow \mathbb{R}^{n_k} \tag{56}$$

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

$$\underbrace{\frac{\partial v_k}{\partial \mathbf{x}}}_{n_k \times n} = \sum_{i=1}^{k-1} \underbrace{\frac{\partial g_k}{\partial v_i}}_{n_k \times n_i} \underbrace{\frac{\partial v_i}{\partial \mathbf{x}}}_{n_i \times n} \tag{57}$$

Therefore, it is this Lipschitz property that might entail how stable an MLP is. We have seen from the universal approximation theorems that for a data set of any size, we can always fit a one-layer perceptron that perfectly fits through all of them, given that the layer is large enough. In these cases, we are interested in fitting the data *smoothly*, and theoretical research in bounding the Lipschitz constant is popular.

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

3 Autograd Engines

In numerical computing packages like `numpy` in Python and `eigen` in C++, we often work with scalars, vectors, and matrices. From linear algebra, the generalization of these objects is a tensor, which is an element of a tensor product space.[3] The full mathematical abstraction is rarely needed in practice, and so developers call tensors by their realization as *multidimensional arrays*.

Definition 3.1 (Tensor)

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

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

1. *Tensor Addition.*
2. *Tensor Additive Inverse.*
3. *Scalar Multiplication.*
4. *Matrix Multiplication.*
5. *Elementwise Multiplication.*
6. *Elementwise Multiplicative Inverse.*
7. *Transpose.*

We would probably like some constructors that allows you to directly initialize tensors filled with 0s (`zeros`), 1s (`ones`), a multiplicative identity (`eye`[4]) Some random initializers would be good, such as sampling from uniforms (`uniform`), `gaussians` (`gaussian`, `randn`).

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

[3]For a refresher, look at my linear algebra notes.
[4]homophone for I, used to denoted the identity matrix.

Algorithm 1 Tensor Class Implementation

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

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

3.1 Strides

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

Definition 3.2 (Stride)

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

$$T[n_1, n_2, \ldots, n_M], \quad 1 \leq n_i \leq N_i \tag{58}$$

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

$$S = (S_1, \ldots, S_M) \tag{59}$$

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

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

$$S_m = \prod_{m<j} N_j, \quad S_M = 1 \tag{60}$$

Example 3.1 (Transposing)

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

$$S = \bigg(\prod_{1<j} N_j, \prod_{2<j} N_j, \ldots, 1 \bigg) \tag{61}$$

to

$$S = \bigg(\prod_{2<j} N_j, \prod_{1<j} N_j, \ldots, 1 \bigg) \tag{62}$$

3.2 Automatic Differentiation

Lemma 3.1 (Derivative of $+/-$)

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

$$\frac{\partial Z_+}{\partial X} = +1 \qquad \frac{\partial Z_+}{\partial Y} = +1 \tag{63}$$

$$\frac{\partial Z_-}{\partial X} = +1 \qquad \frac{\partial Z_-}{\partial Y} = -1 \tag{64}$$

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

Lemma 3.2 (Derivative of Element-wise Multiplication)

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

$$\frac{\partial Z}{\partial X} = Y \qquad \frac{\partial Z}{\partial Y} = X \tag{65}$$

Lemma 3.3 (Derivative of Matrix Multiplication)

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

$$\frac{\partial Z}{\partial X} \in (N, P, N, M) \qquad \left(\frac{\partial Z}{\partial X}\right)_{i,j,k,l} := \frac{\partial Z_{i,j}}{\partial X_{k,l}} \tag{66}$$

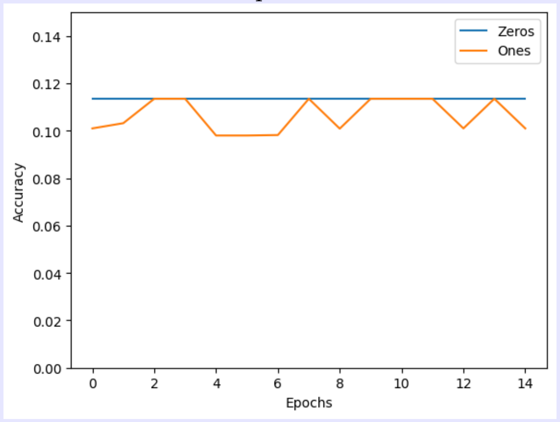
$$\frac{\partial Z}{\partial Y} \in (N, P, M, P) \qquad \left(\frac{\partial Z}{\partial Y}\right)_{i,j,k,l} := \frac{\partial Z_{i,j}}{\partial Y_{k,l}} \tag{67}$$

4 Weight Initialization

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

Example 4.1 (Constant Initialization)

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



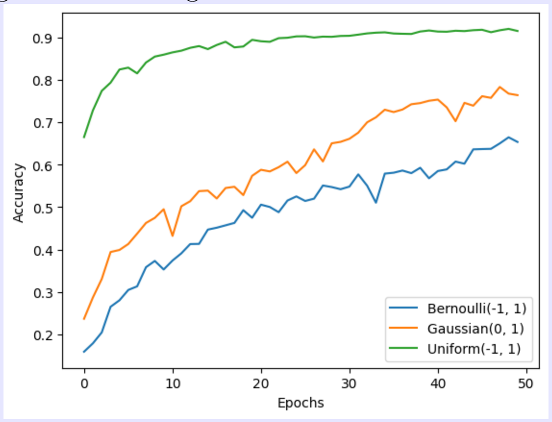
Clearly, this is not good, and theoretically this makes sense since it means all our activations are going to be the same, and thus all our gradients will be the same, meaning that are updates will be the same for every weight, which is not good mixing. We can see this below:

Example 4.2 (Random Initialization with High Variance)

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

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

Running the experiments give the following.

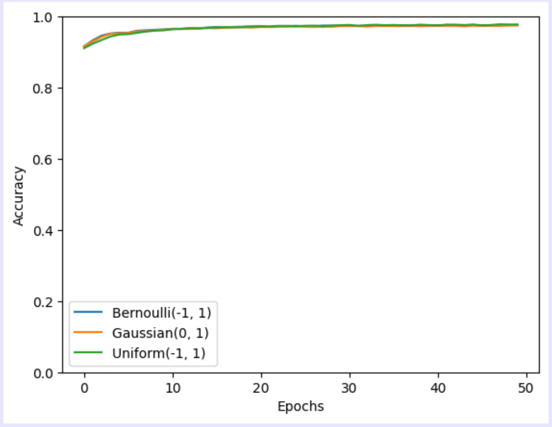


However, this is also not good since it means that the activations will be very large, and thus the gradients will be very large, and so the updates will be very large. This is not good since it means that the weights will be jumping around a lot, and we won't be able to converge. Furthermore, depending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

Example 4.3 (Random Initialization with Low Variance)

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

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



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

Definition 4.1 (Xavier Initialization)

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

$$w_{ij}^{[l]} \sim U\bigg(-\frac{1}{\sqrt{N^{[l-1]}}}, \frac{1}{\sqrt{N^{[l-1]}}}\bigg) \tag{68}$$

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

Code 4.1 (Experimenting with Weight Initializations)

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

5 Activation Functions

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

Definition 5.1 (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
$$f\bigg(\sum_i w_i x_i + b\bigg) \implies \frac{\partial f}{\partial w_i} = f'\bigg(\sum_i w_i x_i + b\bigg)\, x_i \tag{69}$$
which means that the gradients $\nabla_{\mathbf{w}} f$ will always have all positive elements or all negative elements, meaning that we will be restricted to moving in certain nonoptimal directions when updating our parameters.

Definition 5.2 (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.

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

Definition 5.3 (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.

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

Definition 5.4 (Leaky ReLU)

The leaky ReLU

$$\sigma(x) = \max\{0.01x, x\} \tag{70}$$

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

$$\sigma(x) = \max\{\alpha x, x\} \tag{71}$$

Definition 5.5 (ELU)

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

$$\sigma(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\big(\exp x - 1\big) & \text{if } x \leq 0 \end{cases} \tag{72}$$

Definition 5.6 (SELU)

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

$$\sigma(x) = \lambda \begin{cases} x & \text{if } x > 0 \\ \alpha\big(\exp x - 1\big) & \text{if } x \leq 0 \end{cases} \tag{73}$$

where λ and α are constants.

Later on, some further modifications were made, such as the **Swish** and the **Mish** [Mis20] activation functions. These functions have a distinctive negative concavity, unlike ReLU, which accounts for preservation of small negative weights.

Definition 5.7 (Swish)

The Swish activation function is defined as

$$\sigma(x) = x \cdot \sigma(\beta x) \tag{74}$$

where β is a parameter that can be learned.

Definition 5.8 (Mish)

The Mish activation function is defined as

$$\sigma(x) = x \cdot \tanh(\ln(1 + \exp(x))) \tag{75}$$

Code 5.1 (Generating Graphs)

Code used to generate these graphs are here.

6 Popular Benchmark Datasets

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

Definition 6.1 (MNIST and Fashion MNIST)

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

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

Definition 6.2 (CIFAR10 and CIFAR 100)

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

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

Definition 6.3 (ImageNet)

The ImageNet dataset, created at Stanford by Fei-Fei Li [DDS+09], consists of 1.2 million training images and 50k validation images in 1000 classes. This is considered very hard with the basic benchmarks.

Creating your own custom dataset with spreadsheets or images is easy.[5] Loading it to a dataloader that shuffles and outputs minibatches of data is trivial. However, when doing so, you should pay attention to a couple things.

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

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

7 Regularization

7.1 L1 and L2 Regularization

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

Sometimes, it may not always be the best idea to regularize a neural net equally through all weights. For example, weights which may be deeper down the forward pass may focus on more high level features and therefore should be regularized differently than those that are close to the input. Other types of regularization, such as Fiedler regularization [TD20] focuses on preserving the graph structure of the weights.

7.2 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 introduced by [SHK$^+$14], 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: $W^{[l]}_{\text{test}} = p\,W^{[l]}_{\text{train}}$. Another way is to use **inverted dropout**, where we can divide by p in the training stage and keep the testing method the same.

Code 7.1 ()

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

7.3 Layer Normalization

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

Definition 7.1 (Layer Norm)

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

$$x_i \mapsto \frac{x_i - \mathbb{E}[x_i]}{\sqrt{\mathrm{Var}[x_i] + \varepsilon}} \odot \gamma + \beta \tag{76}$$

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

Example 7.1 (Layer Norm)

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

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

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

This also works for higher dimensions.

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

        [[10., 11.],
         [12., 13.],
         [14., 15.],
         [16., 17.],
         [18., 19.]]])
print(ln(x))
tensor([[[-1.5667, -1.2185],
         [-0.8704, -0.5222],
         [-0.1741,  0.1741],
         [ 0.5222,  0.8704],
         [ 1.2185,  1.5667]],

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

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

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

7.4 Batch Normalization

Definition 7.2 (Batch Norm)

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

$$x_i \mapsto \frac{x_i - \mathbb{E}[x_i]}{\sqrt{\mathrm{Var}[x_i] + \varepsilon}} \odot \gamma + \beta \tag{77}$$

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

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

Example 7.2 (Batch Norm 1D)

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

```
>>> bn = nn.BatchNorm1d(5)
>>> x = torch.Tensor(range(10)).reshape(2, 5)
>>> print(x)
tensor([[0., 1., 2., 3., 4.],
        [5., 6., 7., 8., 9.]])
>>> print(bn(x))
tensor([[-1.0000, -1.0000, -1.0000, -1.0000, -1.0000],
        [ 1.0000,  1.0000,  1.0000,  1.0000,  1.0000]],
       grad_fn=<NativeBatchNormBackward0>)
```

For 3D inputs,

```
>>> bn = nn.BatchNorm1d(5)
>>> x = torch.Tensor(range(30)).reshape(2, 5, 3)
>>> print(x)
```

```
tensor([[[ 0.,  1.,  2.],
         [ 3.,  4.,  5.],
         [ 6.,  7.,  8.],
         [ 9., 10., 11.],
         [12., 13., 14.]],

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

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

Example 7.3 (Batch Norm 2D)

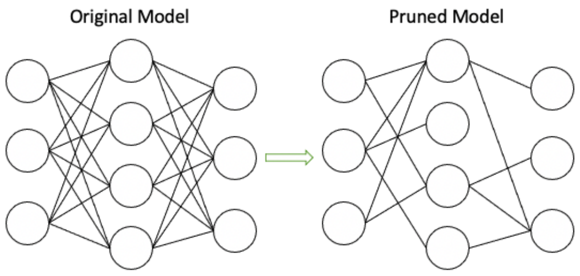
Here is an example of batch norm 2d. There really isn't a difference between these two methods except the dimension that they take in. That is all.

```
>>> bn = nn.BatchNorm2d(5)
>>> x = torch.Tensor(range(60)).reshape(2, 5, 3, 2)
>>> print(x)
tensor([[[[ 0.,  1.],
          [ 2.,  3.],
          [ 4.,  5.]],
          ...
          [58., 59.]]]])
>>> print(bn(x))
tensor([[[[-1.1592, -1.0929],
          [-1.0267, -0.9605],
          ...
          [ 1.0929,  1.1592]]]], grad_fn=<NativeBatchNormBack
ward0>)
```

8 Compression

8.1 Pruning

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



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

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

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

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

This is essentially the **lottery ticket hypothesis** [FC19], which states that a randomly-initialized, dense neural network contains a subnetwork that is initialized such that, when trained in isolation, it can match the test accuracy of the original network after training for at must the same number of iterations.

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

8.2 Quantization

9 Exercises

Exercise 9.1 (Tarokh, ECE685 2021 Midterm 1.1)

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

$$F(x) = \exp\bigg(- \exp\bigg(- \frac{x - \mu}{\beta}\bigg)\bigg) \tag{78}$$

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

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

Solution 9.1

We can derive the PDF of the observation as

$$f(x; \mu, \beta) = \frac{dF(x)}{dx} = \frac{1}{\beta} \exp\bigg\{ - \bigg(\frac{x - \mu}{\beta} + \exp\Big(- \frac{x - \mu}{\beta}\Big)\bigg)\bigg\} \tag{79}$$

and the likelihood is then

$$L(\mu, \beta \mid \mathcal{D}) = \prod_{i=1}^N \frac{1}{\beta} \exp\bigg\{ - \bigg(\frac{x^{(i)} - \mu}{\beta} + \exp\Big(- \frac{x^{(i)} - \mu}{\beta}\Big)\bigg)\bigg\} \tag{80}$$

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

$$\ell(\mu, \beta \mid \mathcal{D}) = -\ln L(\mu, \beta \mid \mathcal{D}) = N \ln \beta + \frac{\sum_i x^{(I)} - N\mu}{\beta} + \sum_{i=1}^N \exp\Big(- \frac{x^{(i)} - \mu}{\beta}\Big) \tag{81}$$

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

$$\frac{\partial \ell}{\partial \mu} = -\frac{N}{\beta} + \frac{1}{\beta} \sum_{i=1}^N \exp\Big(- \frac{x^{(i)} - \mu}{\beta}\Big) \tag{82}$$

$$\frac{\partial \ell}{\partial \beta} = \frac{N}{\beta} - \frac{\sum_i x^{(i)} - N\mu}{\beta^2} + \frac{1}{\beta^2} \sum_{i=1}^N (x^{(i)} - \mu) \exp\Big(- \frac{x^{(i)} - \mu}{\beta}\Big) \tag{83}$$

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

Exercise 9.2 (ECE 685 Fall 2021 Midterm 1.2)

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

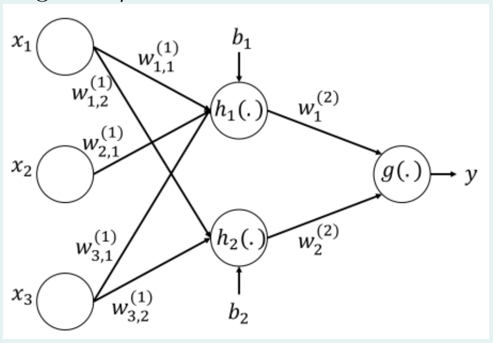
$$\mathbf{w} = (b_1, b_2, w_{11}^{(1)}, w_{12}^{(1)}, w_{21}^{(1)}, w_{31}^{(1)}, w_{32}^{(1)}, w_1^{(2)}, w_2^{(2)}) \tag{84}$$

denote the vector of network parameters.

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

$$L(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^{N} \big(f(x_{1,n}, x_{2,n}, x_{3,n}; \mathbf{w}) - y_n\big)^2 \tag{85}$$

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



Solution 9.2

We can write the computation graph as

$$z_1^{(1)} = w_{11}^{(1)} x_1 + w_{21}^{(1)} x_2 + w_{31}^{(1)} x_3 + b_1 \tag{86}$$
$$z_2^{(1)} = w_{12}^{(1)} x_1 + w_{32}^{(1)} x_3 + b_2 \tag{87}$$
$$a_1^{(1)} = \sigma(z^{(1)}) \tag{88}$$
$$a_2^{(1)} = \tanh(z_2^{(1)}) \tag{89}$$
$$z^{(2)} = w_1^{(2)} a_1^{(1)} + w_2^{(2)} a_2^{(1)} \tag{90}$$
$$y = a^{(2)} = g(z^{(2)}) \tag{91}$$

and composing these gives

$$y = w_1^{(2)} \sigma(w_{11}^{(1)} x_1 + w_{21}^{(1)} x_2 + w_{31}^{(1)} x_3 + b_1) + w_2^{(2)} \tanh(w_{12}^{(1)} x_1 + w_{32}^{(1)} x_3 + b_2) \tag{92}$$

The gradient of the network can be written as

$$\nabla_{\mathbf{w}} L(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^{N} \nabla_{\mathbf{w}} \big(f(x_{1,n}, x_{2,n}, x_{3,n}; \mathbf{w}) - y_n\big)^2 \tag{93}$$
$$= \sum_{n=1}^{N} (f(x_{1,n}, x_{2,n}, x_{3,n}; \mathbf{w}) - y_n) \nabla_{\mathbf{w}} f(x_{1,n}, x_{2,n}, x_{3,n}) \tag{94}$$

where

$$\nabla_{\mathbf{w}} f(x_{1,n}, x_{2,n}, x_{3,n}) = \frac{\partial f}{\partial \mathbf{w}}\bigg|_{\mathbf{x}=\mathbf{x}^{(n)}} \tag{95}$$

Now we can take derivatives using chain rule, working backwards, and using the derivative identities $\sigma'(z) = \sigma(z)(1 - \sigma(z))$ and $\tanh'(z) = 1 - \tanh^2(z)$.

$$\frac{\partial f}{\partial w_1^{(2)}} = \frac{\partial f}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial w_1^{(2)}} = a_1^{(1)} \tag{96}$$

$$\frac{\partial f}{\partial w_2^{(2)}} = \frac{\partial f}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial w_2^{(2)}} = a_2^{(1)} \tag{97}$$

$$\frac{\partial f}{\partial w_{11}^{(1)}} = \frac{\partial f}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial a_1^{(1)}} \frac{\partial a_1^{(1)}}{\partial z_1^{(1)}} \frac{\partial z_1^{(1)}}{\partial w_{11}^{(1)}} = w_1^{(2)} a_1^{(1)} (1 - a_1^{(1)}) x_1 \tag{98}$$

$$\frac{\partial f}{\partial w_{21}^{(1)}} = \frac{\partial f}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial a_1^{(1)}} \frac{\partial a_1^{(1)}}{\partial z_1^{(1)}} \frac{\partial z_1^{(1)}}{\partial w_{21}^{(1)}} = w_1^{(2)} a_1^{(1)} (1 - a_1^{(1)}) x_2 \tag{99}$$

$$\frac{\partial f}{\partial w_{31}^{(1)}} = \frac{\partial f}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial a_1^{(1)}} \frac{\partial a_1^{(1)}}{\partial z_1^{(1)}} \frac{\partial z_1^{(1)}}{\partial w_{31}^{(1)}} = w_1^{(2)} a_1^{(1)} (1 - a_1^{(1)}) x_3 \tag{100}$$

$$\frac{\partial f}{\partial b_1} = \frac{\partial f}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial a_1^{(1)}} \frac{\partial a_1^{(1)}}{\partial z_1^{(1)}} \frac{\partial z_1^{(1)}}{\partial b_1} = w_1^{(2)} a_1^{(1)} (1 - a_1^{(1)}) \tag{101}$$

$$\frac{\partial f}{\partial w_{12}^{(1)}} = \frac{\partial f}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial a_2^{(1)}} \frac{\partial a_2^{(1)}}{\partial z_2^{(1)}} \frac{\partial z_2^{(1)}}{\partial w_{12}^{(1)}} = w_2^{(2)} (1 - (a_2^{(1)})^2) x_1 \tag{102}$$

$$\frac{\partial f}{\partial w_{13}^{(1)}} = \frac{\partial f}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial a_2^{(1)}} \frac{\partial a_2^{(1)}}{\partial z_2^{(1)}} \frac{\partial z_2^{(1)}}{\partial w_{13}^{(1)}} = w_2^{(2)} (1 - (a_2^{(1)})^2) x_3 \tag{103}$$

$$\frac{\partial f}{\partial b_2} = \frac{\partial f}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial a_2^{(1)}} \frac{\partial a_2^{(1)}}{\partial z_2^{(1)}} \frac{\partial z_2^{(1)}}{\partial b_2} = w_2^{(2)} (1 - (a_2^{(1)})^2) \tag{104}$$

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

$$\nabla_{\mathbf{w};\mathcal{M}} L(\mathbf{w}) = \sum_{(\mathbf{x},y) \in \mathcal{M}} (f(\mathbf{x};\mathbf{w}) - y) \nabla_{\mathbf{w}} f(\mathbf{x}) \tag{105}$$

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

$$\mathbf{w} = \mathbf{w} - \eta \nabla_{\mathbf{w};\mathcal{M}} L(\mathbf{w}) \tag{106}$$

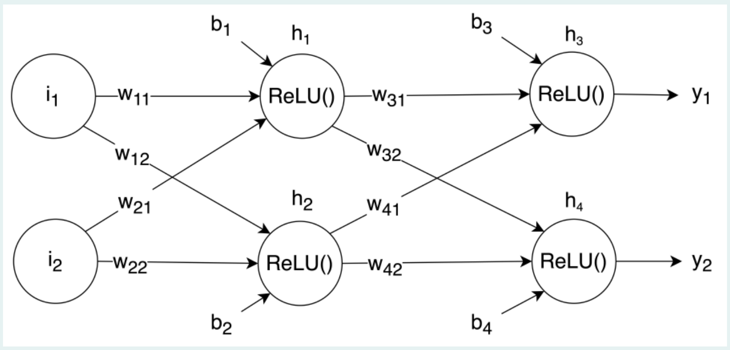
Exercise 9.3 (ECE 685 Fall 2021 Midterm 1.3)

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

$$h_1 = \text{ReLU}(x_1 w_{11} + x_2 w_{21} + b_1) \tag{107}$$

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

$$y_1 = \text{ReLU}(h_1 w_{31} + h_2 w_{41} + b_3) \tag{108}$$



The values of the variables are given as follows:

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

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

References

[CLHN93] An Mei Chen, Haw-minn Lu, and Robert Hecht-Nielsen. On the geometry of feedforward neural network error surfaces. *Neural Computation*, 5(6):910–927, 11 1993.

[DDS$^+$09] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[FC19] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks, 2019.

[Mis20] Diganta Misra. Mish: A self regularized non-monotonic activation function, 2020.

[SHK$^+$14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

[TD20] Edric Tam and David Dunson. Fiedler regularization: Learning neural networks with graph sparsity, 2020.