Contents

6.1 Modelling XOR

6.11 Working with the normal equation of Linear Regression with bias term (page 172)

In a separate file, called proof of normal equation, there's a derivation of the normal equation using the differential approach to make life easier. On page 172, we have almost the same situation except there's a bias term in the linear regression model, so we have to account for that. I tried to solve this with pen and paper by letting the bias term be a separate vector and compute the optimal w and b, but it didn't seem to work. I found out that you need to fuse the bias term into the X matrix by adding a column of ones and adding one extra entry in the weight vector w as noted by Goodfellow. This emulates the effect of adding the bias term to each data point. Just to clarify the way to interpret the $X^{(\text{train})}$ dataset is the columns are features and the rows are data points. For $y^{(\text{train})}$ this is the labels for $X^{(\text{train})}$, so for instance row one in $y^{(\text{train})}$ is the label for data point one on row one in $X^{(\text{train})}$.

In our case, if we add a column of ones to the existing training dataset, we get:

$$X^{(\text{train})} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

while the labels remain the same:

$$y^{(\text{train})} = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

The weight vector w will have an extra bias term

$$w = \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix}$$

where w_3 is the bias term now.

Because we already have the normal equation from chapter 5, we can reuse it to compute the optimal w that includes b as well. Recall that the normal equation is:

$$w = \left(X^{(\text{train})^T} X^{\text{train}}\right)^{-1} X^{\text{train}} y^{\text{train}}$$

Inputting the values into the equation gives:

$$w = \begin{pmatrix} 0 \\ 0 \\ \frac{1}{2} \end{pmatrix}$$

where the first two entries constitute the actual weights for the data w_{actual}, and the last entry is the bias term b, which remains the same for all data points. This explains how the author got $w = 0$ and $b = \frac{1}{2}$ as the optimal values.

To compute these values, I used the following code in a Jupiter notebook and matched against the provided answer in the book:

```python
import numpy as np
from numpy.linalg import inv
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

# basically just computing w = (X^TX)^-1 (Xy)

x = np.array([[0, 0, 1], [0, 1, 1], [1, 0, 1], [1, 1, 1]])
y = np.array([0, 1, 1, 0])
x.T @ x
xinv = inv(x.T @ x)
xinv
w = xinv @ (x.T @ y)
w
# Check that the computation is the same as the solution in the book
np.allclose(np.array([0, 0, 0.5]), w)

"""
Output:

array([[2, 1, 2], [1, 2, 2], [2, 2, 4]])

array([[ 1.00000000e+00, -1.11022302e-16, -5.00000000e-01], [ 0.00000000e+00,
1.00000000e+00, -5.00000000e-01], [-5.00000000e-01, -5.00000000e-01,
7.50000000e-01]])

array([0.00000000e+00, 2.22044605e-16, 5.00000000e-01])

True
"""
```

6.12 Understanding the left figure 6.1 page 173

Intuitively, it's easy to see that a straight line cannot perfectly separate the left figure, however the informal explanation given by Goodfellow made me a bit confused, so I tried to break it down. Here's the quote:

> When $x_1 = 0$, the model's output must increase as x_2 increases. When $x_1 = 1$, the model's output must decrease as x_2 increases. A linear model must apply a fixed coefficient w_2 to x_2. The linear model therefore cannot use the value of x_1 to change the coefficient on x_2 and cannot solve this problem.

We have the linear model

$$f(x) = w_1 x_1 + w_2 x_2 + b$$

and we have these situations $x_1 = 0$ and $x_1 = 1$ to look at. Remember that our goal is to get the function to output 0 when both input values are the same and 1 when exactly one of them is 1.

1. **When $x_1 = 0$: The model's output must increase as x_2 increases.**
   - In this case, we are looking at points $(0, 0)$ and $(0, 1)$ in the x-space.
   - For a linear model, the output can be described as $f(x) = w_1 x_1 + w_2 x_2 + b$.
   - When $x_1 = 0$, this reduces to $f(x) = w_2 x_2 + b$.

    - To correctly classify $(0,0)$ as 0 and $(0,1)$ as 1, the model's output needs to increase when x_2 changes from 0 to 1.
    - This means w_2 should be positive so that as x_2 increases, the output increases.
2. **When $x_1 = 1$: The model's output must decrease as x_2 increases.**
    - Now we are looking at points $(1,0)$ and $(1,1)$ in the x-space.
    - For a linear model, the output is $f(x) = w_1 x_1 + w_2 x_2 + b$.
    - When $x_1 = 1$, this becomes $f(x) = w_1 + w_2 x_2 + b$.
    - To correctly classify $(1,0)$ as 1 and $(1,1)$ as 0, the model's output needs to decrease when x_2 changes from 0 to 1.
    - This means w_2 should be negative so that as x_2 increases, the output decreases.

However, w_2 is fixed for all data points, and can't be both negative and positive, it can only be one of them. Therefore, it's impossible for the linear model to learn the XOR function.

6.13 The need for non-linearity

As we already know the reason we need non-linearity is so that the layers don't collapse into one single linear layer. Recall that any linear transformation mapping finite-dimensional vector space to another finite-dimensional vector space $f : V \longrightarrow W$ can be represented by a standard matrix A

$$f(x) = Ax.$$

The exact meaning of vector space is not important for our purposes, just think of it as a set of vectors or matrices that act like we expect them to (fulfilling certain properties). This means that if we were to stack fully connected (every neuron has a weight to the features of the data) linear layers on top of each other in a neural network then we get

$$f^n \circ f^{n-1} \circ \dots \circ f^1(x) = A^{(n)} A^{(n-1)} \dots A^{(1)} x = \left(A^{(n)} A^{(n-1)} \dots A^{(1)}\right) x = Bx,$$

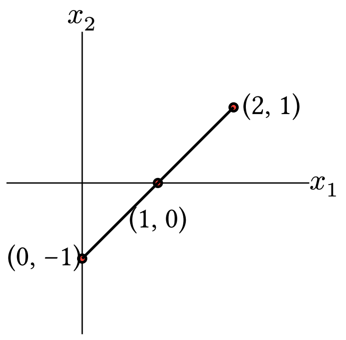
which just transforms into another matrix B showing the collapse of the layers. This means that there would be no use in having n layers, because the network can just be represented by one single linear layer instead. If we however add non-linearity to it, we can no longer collapse them, so that the layers may provide some value now.

6.14 Equation 6.9 page 176 - linear line passing through the points and explaining the outputs

Each row represents the transformed point after the first linear layer before the activation function

$$XW + c = \begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

So write them out in a 2D coordinate system, then you will see that they all indeed lie on a linear line.



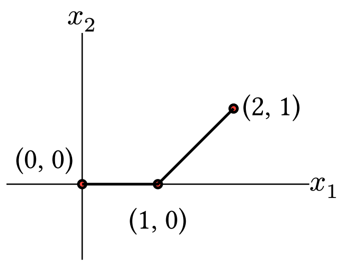
Function outputs for the data points are

x_1	x_2	**XOR Output**
0	−1	0
1	0	1
2	1	0

This means that the function needs to increase from 0 to 1 and then drop back down to 0 again, which is not possible for a linear function (a straight line or plane can't do this) according to the author. However, I think it's more about the fact that following along the line above, the x_1 and x_2 input will increase steadily. Therefore, there's no way to set the weights such that the model will increase and then decrease after a certain point, because once you set the weights so that the model increases then it will keep increasing or if set so that it decreases it will keep decreasing. So the problem is not really that the function needs to increase and then decrease, which we will see in the later layer that it's possible to achieve, it's more because of where the points are located that is causing the issue. Now applying relu on the transformed data gives

$$\mathrm{relu}(XW + c) = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

And writing this in the 2D coordinate system



This means the output function needs to be

x_1	x_2	**XOR Output**
0	0	0
1	0	1
2	1	0

Here the model needs to increase and then decrease again. We can see that (and probably the only solution given the weights we have and biases) if x_1 increases (set weight w_1 positive), while x_2

decreases (set weight w_2 negative), then the model will be able to increase from 0 to 1 and then decrease again from 1 to 0. The intuition is because x_1 is always increasing in this small interval going from 0 to 1 to 2, while x_2 is non-decreasing going from 0 to 0 and then to 1. Going from $(0, 0)$ to $(1, 0)$ means that x_2 has no influence on the output since

$$f = w_1 \cdot 1 + w_2 \cdot 0 + b = w_1 + b$$

so we need to make x_1 increase by setting the weight w_1 positive. Now going from $(1, 0)$ to $(2, 1)$ means we have

$$f = w_1 \cdot 2 + w_2 \cdot 1 + b = 2w_1 + w_2 + b$$

so both inputs have influence now. How do we go from output 1 to output 0? We need to decrease the model output. Since $w_1 > 0$ is positive, then the only way to decrease the model is to decrease x_2 by setting $w_2 < 0$ negative.

As for the format of the model output what we want is the XOR output for each data point we had. We had 4 data points, so we expect to get 4 elements in a vector as the final output from the model, telling us the XOR evaluation of each data point. Thus multiplying the output from the first layer with the weight matrix

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

gives

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

which is correct according to the XOR truth table above.

Lastly, geometrically I think one can interpret the final layer as a plane, and if we were to illustrate the input and output relation between the data input and the model output then it would be in a 3D coordinate system. This means that if the plane can't contain all the data points in the coordinate system, then it can't model the XOR function for this particular dataset. Note that the author said the model wouldn't be able to generalize, therefore it only works on this small dataset because it's overfitting to it.

6.2 Gradient-Based Learning

To avoid clutter I've moved the long proofs to a file called Background, they mostly deal with things that weren't trivial to prove.

6.21 What does semicolon mean in functions?

At least in deep learning, often in parametrized models when representing them as functions we see something like

$$f(x_1, x_2, ...; p_1, p_2, ...)$$

which just means that the model, once supplied with the parameters $(p_1, p_2, ...)$, will produce a function that accepts the variables $(x_1, x_2, ...)$. So the general pattern is

$$f(\text{variables}; \text{parameters})$$

As an example consider a univariate gaussian distributed model that we denote as

$$y = f(x;\theta) = f(x;\mu,\sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} \exp^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

This is something that should be familiar, it's just a gaussian distribution with one variable! Here x is the input variable and μ and θ are the model parameters. Now if we assume that the model has been trained, then the parameters will constitute the trained parameters, so just substitute them into the formula and you now have an actual model that will produce outputs for you given input x.

Another interpretation, **that I'm not too sure of**, is when you equate it to a conditional distribution meaning

$$f(x;\theta) = f(y \mid x,\theta)$$

where you view the parameters and the data x as random variables and the model produces not a single output, but a probability distribution, in this case a conditional probability distribution. Specifically, in practise it provides the parameters to the probability distribution that the user has to choose beforehand. Then in inference time, depending on the task but let's assume that we modelled a gaussian distribution for the entire data, when you give an input x it outputs y. I guess this is more of a probabilistic view, perhaps even Bayesian? I'm not sure. This does come up in this section a lot though and I often just view it as a probability distribution with parameters θ and input variable x. **(Not sure about this paragraph)** The bigger point here is to not view it as a literal conditional probability distribution

$$p(y \mid x) = \frac{p(x,y)}{p(x)}$$

as that would give a different form on the probability distribution I suppose, rather view it as a probability distribution that has θ provided as parameters and that accepts inputs x to produce output y.

6.22 Learning the parameters of a distribution instead of a direct prediction

This is mentioned on page 188, which I thought was worth bringing up. When you define a general distribution to model you can write it as

$$p(y \mid x;\theta)$$

and applying MLE would give the cost function

$$-\log p(y \mid x;\theta)$$

If however the network predicts the parameters ω to the probability distribution over y and not the actual value y, we would get

$$f(x;\theta) = \omega$$

and the cost function changes to

$$-\log p(y;\omega(x)) = -\log p(y \mid x;\omega(x))$$

So the main idea is that the function $f(x;\theta) = \omega$ and not $f(x;\theta) = y$ anymore, meaning it's predicting the parameters of the distribution rather than the output value y of the distribution.

6.23 Gradient descent learning intro

Stochastic gradient descent is sensitive to initial parameters and the same goes for mini-batch gradient descent. For large data GD is almost always the choice for neural networks and common for traditional ML models as well. There are lots of initialization tactics, but these will be addressed in chapter 8, for now they propose to initialize weights to random small numbers and the bias to either zero or small positive values without motivation.

Negative log likelihood is good because it cancel exponentiation, where the saturation can occur for various activation functions if the input is very negative. It's also nice to work with as you directly obtain a cost function. But watch out for cases when the model can control the density of output distribution, because in that case the model can cheat and reap unlimited reward by assigning extremely high density to the correct training set outputs making the cross-entropy infinitely negative. This is possible because cross-entropy doesn't have a minimum value. To combat this use regularization.

Mean squared error and absolute error not good for gradient based optimization according to the author, can't negate saturations in activation functions, which leads to small gradients. Despite this, in practise I do see mean squared error used often, so I think there are probably some remedies like regularization or inherent nature of architecture that still makes it doable to train neural networks with mean squared error.

6.24 Binary and categorical outputs - output units

Choice of cost function heavily influenced by choice of output unit. Usually use cross-entropy as cost function, but the exact form of cross-entropy will also change depending on what output unit and hidden unit is chosen. Output unit seems to be activation function applied on the layer output, and hidden unit is just the activation function.

6.241 Linear unit + gaussian output distribution

If there is no non-linearity then given a linear layer that predicts the mean of a conditional gaussian (conditioned on data and fixed variance), the MLE on it will be equivalent of a mean squared error. However, linear units are not good to use for learning parameters if they have certain constraints, depending on how complicated those constraints are. You usually use other output units for those cases or wrap it around an activation function that can better handle constraints. Otherwise, using linear units is easy as they don't cause issues like saturation etc.

6.242 Sigmoid units for Bernoulli output distribution

This is for binary settings, only two kinds of outputs. The MLE approach is to define a Bernoulli distribution over y conditioned on x. The usual way would be to set up the Bernoulli probability function and apply negative log likelihood on it like below

$$P(y, h) = -[y \log(h) + (1 - y) \log(1 - h)]$$

where h(z) is the output from a layer. We are only considering one layer at the moment. The author mentions that for numerical stability it is usual to define the negative log-likelihood as a function of h in this case, rather than as a function of y. For things like sigmoid if $h(z)$ is sigmoid, then we would get a more numerical stable expression log exp sum $\log(1 + \exp^{-z})$, which is more stable despite $\exp^{\text{big negative number}}$ moving towards zero for big negative values, since we always add 1 at the end. It's important to note that even though this is more **numerical stable**, it still doesn't solve the issue with **saturation** for sigmoids, as this will give 0 or near zero gradients for big negative values.

Going back to the original problem, in this particular case the restriction for the output needs to be within $[0, 1]$, which is not possible with just a linear layer, because of problems with saturation, look at eq 6.18 in the book. The author therefore proposes a sigmoid activation function to achieve this. This needs some further refinement by exponentiating and normalizing the output from it. A non-trivial step was shown between eq 6.22 and 6.23 so I will provide it here. Write it out as a piece-wise function

$$\frac{\exp(yz)}{\sum_{y'=0}^{1} \exp(y'z)} = \begin{cases} \sigma(z) \text{ if } y = 1 \\ \sigma(-z) \text{ if } y = 0 \end{cases}$$

Now we can use trial and error to find a suitable expression that will result in these cases, when $y = 1 \Rightarrow z$ and $y = 0 \Rightarrow -z$. The easiest expression that fulfills this is $(2yz - z) = (2y - 1)z$, which therefore leads to eq 6.23 $\sigma((2y - 1)z)$. Applying what we have learnt, that MLE gives a cross-entropy as cost function, we therefore get eq 6.24

$$J(\theta) = -\log P(y|x) = -\log \sigma((2y - 1)z) = \{\text{use } \sigma(z) = -\varsigma(-z)\} = \varsigma((1 - 2y)z)$$

where ς is the softplus function

$$\varsigma = \log(1 + \exp^{z})$$

which is very similar to relu but is differentiable at all points, but also mathematically is similar to sigmoid in that it can be written in terms of sigmoid as we have just seen.

The interpretation here is that log undoes the exp in the numerator, thus preventing some issues with saturation in the output from the sigmoid function. It still saturates, but only when it's very negative and therefore only when the prediction is correct $y = 1, z > 0$ and $y = 0, z < 0$. For wrong signs on z we get $|z|$, which implies that we get as derivative of the absolute value $\frac{z}{|z|} = \text{sign}(z)$, meaning the gradient will be large enough to correct mistakes made by the model predictions. The reason we get $|z|$ for incorrect signs on z is because when z is a different sign than what (1-2y) evaluates to, given that y can only be 0 or 1, we will always have a factor of 1 or −1 from (1-2y), which doesn't change the value of z because $1 \cdot z = z, -1 \cdot z = -z$, but since the author here is talking about the case when z and (1-2y) are different signs we always get a positive value back. Therefore we can simplify this to just |z|.

Compared to mean squared error the cost function would saturate for both big negative values and positive if we use a sigmoid like this. This means it can saturate for both correct and incorrect predictions, which is not good because it can kill or slow down the gradient learning process considerably.

6.243 Softmax units for multinoulli output distribution

Note that multinoulli is another name for multinomial which just means categorical, so all output events need to sum to 1 and there are more than 2 outputs.

For multiple outputs use softmax

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

With cross-entropy we get the following cost function

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

The author explains this as the first term pushing z_i up while the second term pushes all z down, because it will always be negative so it will always subtract from any given z_i. One interpretation of this expression we can get is if we consider the case when $\max_j z_j \gg z_k, k \neq j$, that is the max z value is considerably bigger than other values in z vector, then the second term will be dominated by this element, which is why it evaluates to approximately

$$\log\left(\exp\left(\max_j z_j\right)\right) = \max_j z_j$$

The intuition here is that if the correct answer has a large input compared to the other elements, then the first term and second term will cancel each other

$$z_i - \max_j z_j \approx z_i - z_i = 0$$

so the rest of the loss in the cost function will come from the incorrectly classified inputs, which is desirably because adding more loss to an already correctly classified input is not useful.

When the dataset gets larger and larger the softmax will converge to approximately the true fractions of each outcome in the dataset.

Loss functions that want to use softmax but don't use log will not undo the exp and thus will not be able to handle cases when the input is very negative, because that will lead to vanishing gradients $\exp(\text{big negative}) \approx 0$.

To make softmax numerical stable a common method is to subtract each input with the max of all the inputs, which is fine since softmax is invariant under scalar addition. The key idea is since every element will be followed by an added scalar now in the exponent, we can simply factor this scalar out in both the numerator and denominator, and then cancel them out, so it will be almost as if they didn't exist

$$\frac{\exp(z+c)}{\sum_j \exp\left(z_j + c\right)} = \frac{\exp(c)\exp(z)}{\exp(c)\sum_j \exp\left(z_j\right)} = \frac{\exp(z)}{\sum_j \exp\left(z_j\right)}$$

So this seems pretty pointless, we haven't achieved anything? Not quite, this is more numerical stable because the summation in the denominator will always have at least 1 in the exponent in the denominator $\exp(0) = 1$ because at least one must be $\max_i x_i$ meaning $x_j - \max_i x_i = 0$, where $x_j = \max_i x_i$. So in the case of saturation when we have a lot of negative values $\exp(\text{big negative}) \approx 0$, this term will add $\exp(0) = 1$ to all other zeros or near zeros, which will equate to approximately 1, preventing division with zero error or division with very small values underflow issues.

In this new formulation of softmax saturation can still occur, when $\max_i z_i \gg z_j, j \neq i$ and the input z_i is the maximum, then it will saturate to 1 which we can see here

$$\operatorname{softmax}(z)_i = \frac{\exp(z_i - \max_i x_i)}{\sum_j \exp\left(z_j - \max_i z_i\right)} \approx \frac{\exp(0)}{\exp(0)} = 1$$

If $\max_j z_j \gg z_i, j \neq i$ where z_i is NOT maximum, then the softmax will saturate to 0

$$\operatorname{softmax}(z)_i = \frac{\exp\left(z_i - \max_j z_j\right)}{\sum_j \exp\left(z_j - \max_j z_j\right)} \approx \frac{\exp(\text{big negative})}{\sum_j \exp\left(z_j - \max_j z_j\right)} \approx 0$$

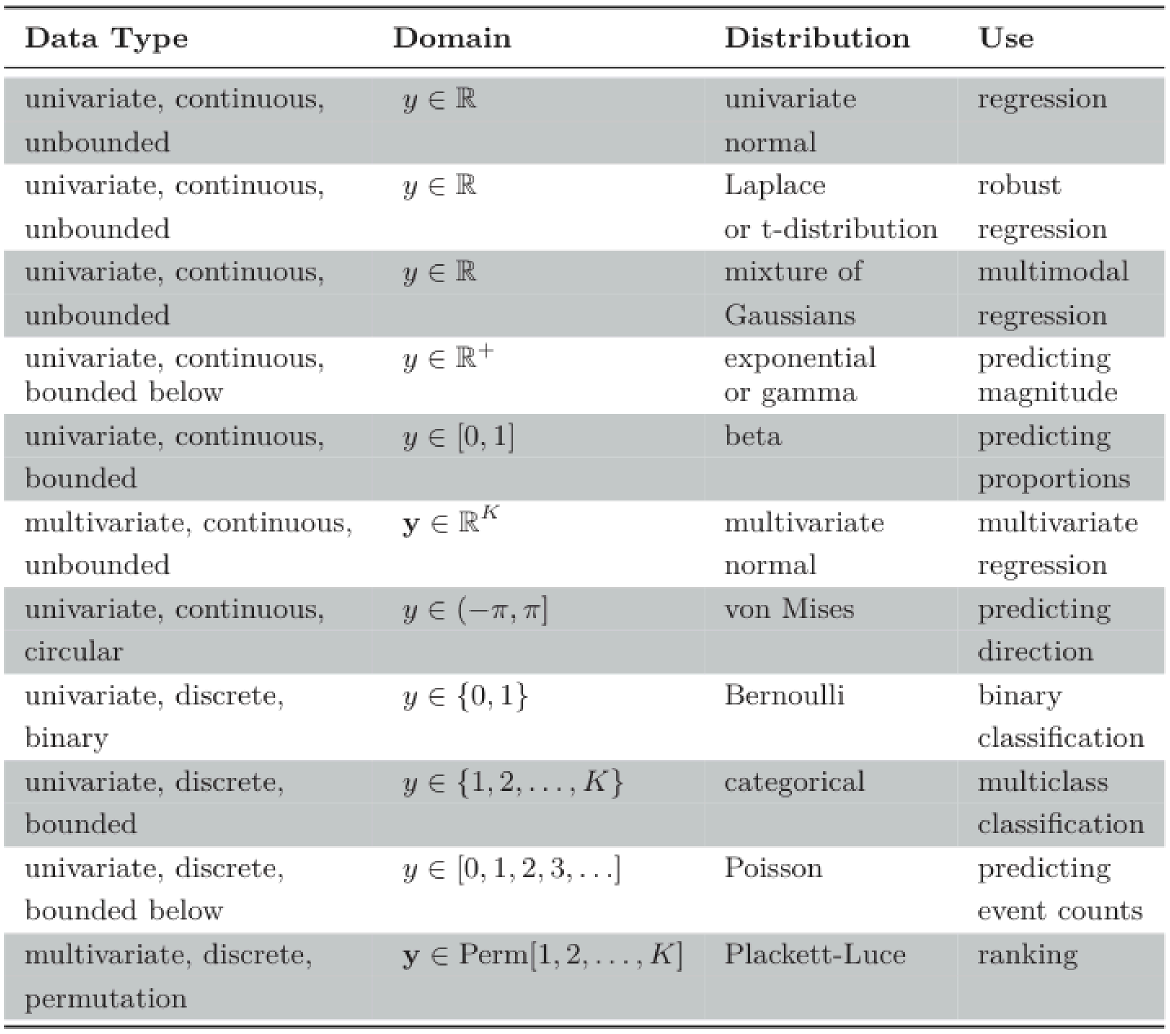
This means that if the cost function doesn't account for the exp by applying log, then they will face the same difficulties of dealing with saturation issues.

The author mentioned something about being parameter efficient with softmax, but in practise it seems that it doesn't matter, and since the less parameter efficient version is easier to implement, just go with it instead.

Lastly, the dynamics of the outputs from layers going into softmax creates a competitive environment between the elements, where it creates a "the winner takes it all" situation in worst case. This is because of the restriction that all normalized outputs must sum to 1, so outputs that gets assigned more weight (difference between max element and the others is big) will naturally take up more probability density/mass, and at the extreme cases will move everything else to 0 while it will move towards 1. This is a known issue in for instance self-attention, where it's a must to rescale the outputs by normalizing them by their magnitude before going into the softmax.

6.244 Other output types
Don't understand the reasoning on training covariance matrix in gaussian.

Taken from prince here's an overview of different output types in Figure 1



Data Type	Domain	Distribution	Use
univariate, continuous, unbounded	$y \in \mathbb{R}$	univariate normal	regression
univariate, continuous, unbounded	$y \in \mathbb{R}$	Laplace or t-distribution	robust regression
univariate, continuous, unbounded	$y \in \mathbb{R}$	mixture of Gaussians	multimodal regression
univariate, continuous, bounded below	$y \in \mathbb{R}^+$	exponential or gamma	predicting magnitude
univariate, continuous, bounded	$y \in [0, 1]$	beta	predicting proportions
multivariate, continuous, unbounded	$\mathbf{y} \in \mathbb{R}^K$	multivariate normal	multivariate regression
univariate, continuous, circular	$y \in (-\pi, \pi]$	von Mises	predicting direction
univariate, discrete, binary	$y \in \{0, 1\}$	Bernoulli	binary classification
univariate, discrete, bounded	$y \in \{1, 2, \ldots, K\}$	categorical	multiclass classification
univariate, discrete, bounded below	$y \in [0, 1, 2, 3, \ldots]$	Poisson	predicting event counts
multivariate, discrete, permutation	$\mathbf{y} \in \text{Perm}[1, 2, \ldots, K]$	Plackett-Luce	ranking

Figure 1: Output types.

6.3 Hidden Units

Non-differentiability in activation functions is in practise usually not a problem, as there are very few points these occur at so it's unlikely that outputs will be at exactly these points as there will always be rounding errors. Moreover, libraries usually will use the direction of derivative that is defined even though some other directions aren't from the same point or output 0 if non of the directions are differentiable. Also it's rare for the training process to reach a local minium of the cost function, instead the most important work of the training should be to move it as close as possible. Therefore, non-differentiability isn't an issue in training usually.

Relu is the most popular choice, it's easy to compute and behaves like an identity function for positive values, so it doesn't saturate, only at negative values. There are other variants to deal with this shortcoming such as leaky relu, but in practise they seem to perform comparable. One practical advice is to initialize bias to small positive values such as 0.1 if using a linear layer, to get the training going at the start.

Tanh better option than relu, is centered and thus resembles the identity function more closely than sigmoid. It can be expressed in terms of sigmoid as

$$\tanh(z) = 2\sigma(2z) - 1$$

the proof of which is not obvious, but I will add it in the Background.typ file to avoid clutter. In Figure 2 we can see a comparison of sigmoid and tanh

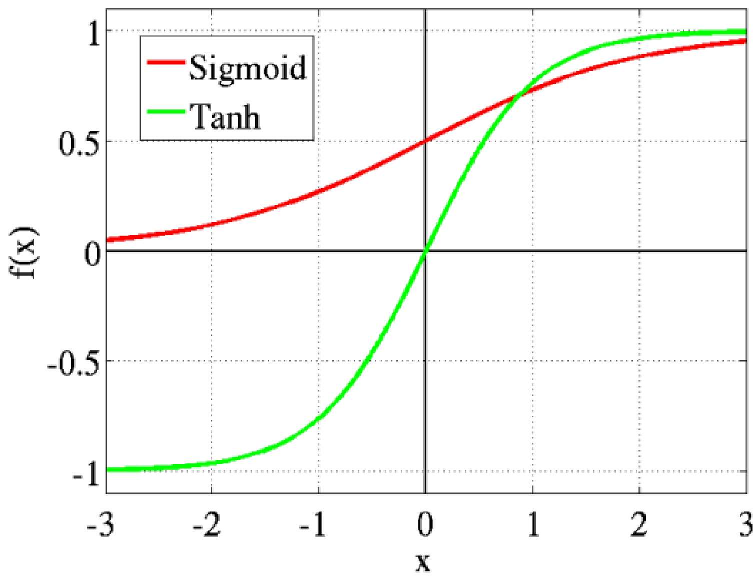


Figure 2: Tanh vs sigmoid.

If the activations in tanh can be kept small, then it resembles training linear model. Compare this

$$y = w^T \tanh(U^T \tanh(V^T x))$$

to

$$y = w^T U^T V^T x$$

One thing that was surprising to me was the fact that several linear layers on top of each other can have benefits in that it can reduce the number of parameters. Let's assume we have

- n inputs
- p outputs
- U is $n \times q$
- V is $q \times p$

then if the model is defined as

$$V^T U^T x$$

we will have a total of $(q \cdot n) + (p \cdot q) = q(p + n)$ parameters. If we instead just had one big matrix to represent this linear model with w of size $n \times p$ we would have np parameters. This means if q can be kept small, then we require fewer parameters than the matrix w. As a consequence of this it requires U and V to be low rank matrices. This shows that stacking linear layers on top of each other can have some benefit if parameter saving is important.

Softplus is discouraged as an activation function, even though it theoretically appears to be a better version of relu. Goes to show the counterintuitive nature of deep learning sometimes.

6.4 Architecture Design

6.41 Universal approximation theorem

> A feedforward network with a linear output layer and at least one hidden layer with any "squashing" activation function (such as the logistic sigmoid activation function) can approximate any Borel measurable function from one finite-dimensional space to another with any desired non-zero amount of error, provided that the network is given enough hidden units
>
> — Page 198

Derivatives of feedforward NN can also approximate the derivatives of the original function arbitrarily well in theory.

But this is just in theory, in practise it can be hard to find the correct parameters to model a function as close as possible, for instance gradient-based learning can go wrong because the loss landscape doesn't behave nicely etc. Another reason is overfitting might lead to trapping the model into a local minima that it can't escape, so it settles at a less optimal function.

6.42 Bounds on the size of single-layer nn

In worst case it's exponential, but I thought it would be instructive to at least explain the binary example. It's based on the rule of product that for most people comes naturally without thinking about it. I think a disjoint tree of choices is best way to illustrate the idea in Figure 3

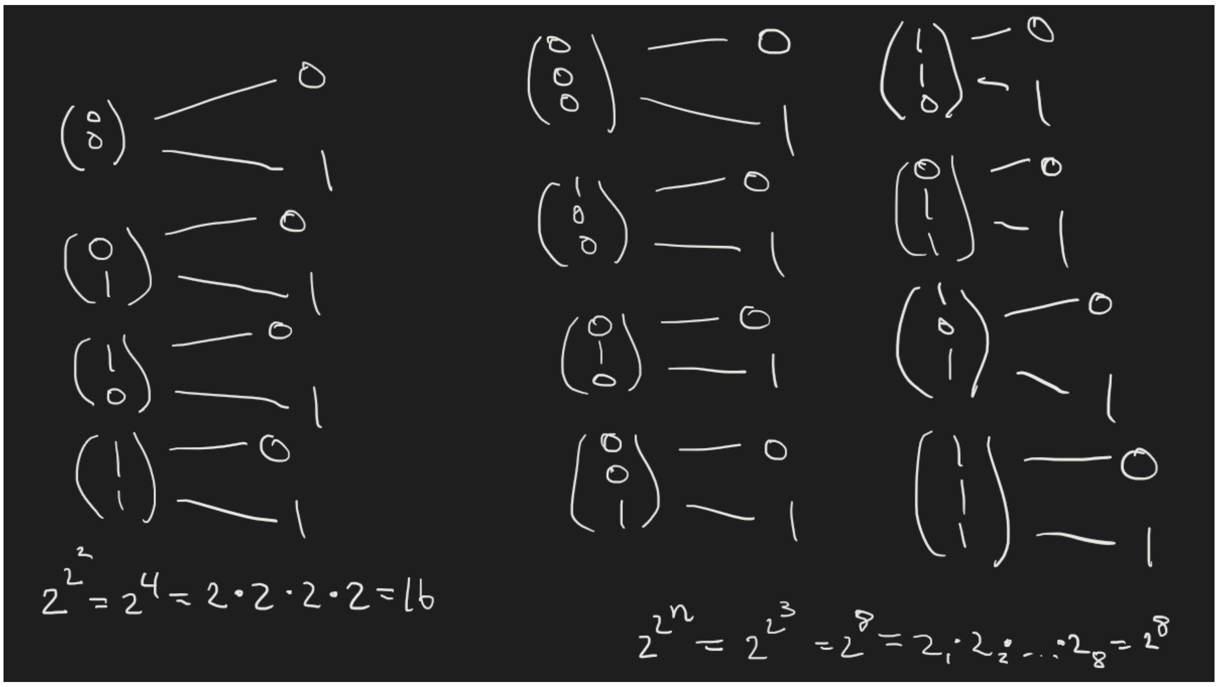

$2^{2^2} = 2^4 = 2 \cdot 2 \cdot 2 \cdot 2 = 16$
$2^{2^n} = 2^{2^3} = 2^8 = 2_1 \cdot 2_2 \cdot \ldots \cdot 2_8 = 2^8$

Figure 3: All possible unique binary functions for a given set of binaries.

The idea is each tree will contribute a factor to the total number of possible unique binary function mapping. The left side is the inputs and the right side are the possible outputs for that input. The rule of product says that if we have a, b choices, then the total number of choices is $a \cdot b$. If we have for instance $v \in \{0, 1\}^2$, then there are 2^2 different inputs and each such input can lead to two different mappings 0 or 1, so there are according to the image above $2 \cdot 2 \cdot 2 \cdot 2 = 2^{(2^2)} = 2^4 = 16$ possible unique mappings one can construct.

6.43 Neuron connections with weights illustration

This is an example of a fully-connected neural network with one single layer that consists of 2 hidden units. It's just to explicitly show where exactly the weights are in the neural network, a visual connection to the weight matrix we always see mentioned in the formulas. We can see in the first row of the weight matrix that it constitutes the weights for the first neuron in the hidden layer, $w_{11}, w_{12}, w_{13}, w_{14}$, while the second neuron has weights on the second row of the weight matrix illustrated in green. In a fully-connected layer, each neuron has a connection via a weight to each of the input features.

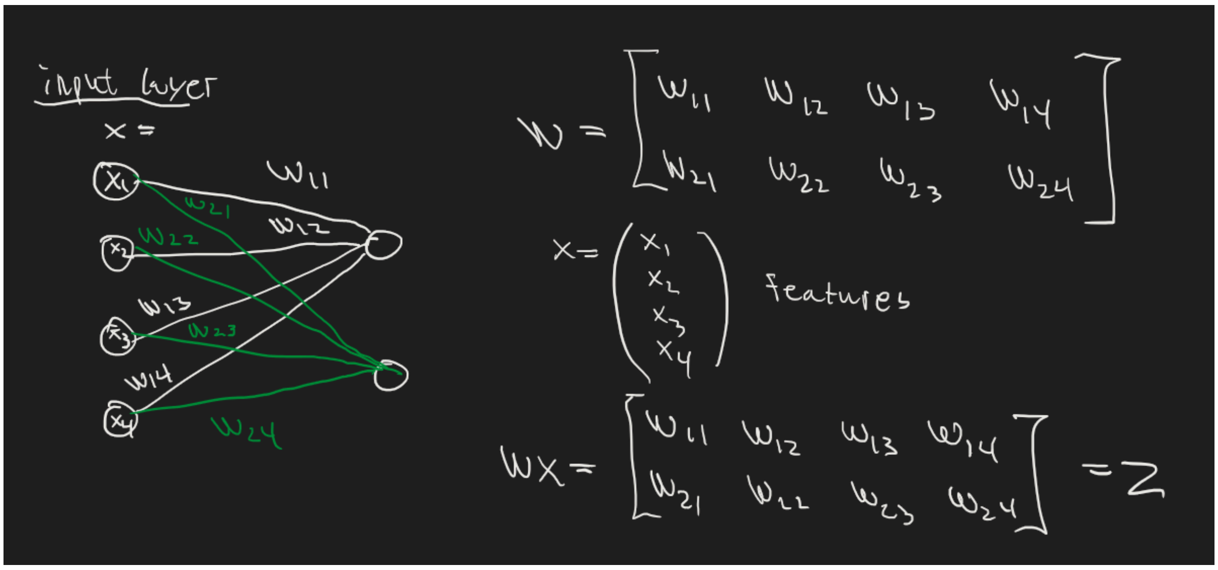

input layer
x =
x_1
x_2
x_3
x_4
w_{11}
w_{21}
w_{22}
w_{12}
w_{13}
w_{23}
w_{14}
w_{24}
$W = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \end{bmatrix}$
$x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix}$ features
$Wx = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \end{bmatrix} = z$

Figure 4: Neuron connections with weights

6.44 Explaining figure 6.5

The hyperplane is given by $wx = z$, where w is a weight vector and x the input vector, z is the output. Because we have an absolute relu activation $|z|$ applied on the output of the hyperplane we will only get positive values, this will be what we see on an absolute function, where we get a mirror on both sides of an axis. The axis in our case is the hyperplane when $z = 0$. Because they are just mirrors of each other, we can just decide to use just one part of the side, which emulates the folding mechanism the author is talking about. Now in the next layer when we again apply a linear transformation and non-linear activation function on the output from the previous layer, then we get another fold and so on and so forth.

6.45 Very informal notes

- exists family of functions that **can** be approximated efficiently by a deep architecture (many many layers) over a certain d number of layers. Shallower less than d will need exponential number of hidden units to achieve the same thing that deeper networks can.
- nn with relu has demonstrated to possess the universal approximation properties, although the exact width and efficiency is not specified, only that a sufficiently wide nn could represent any function
- eq 6.42 and 6.43, the author says there's no guarantee that the functions (the actual functions we want to ideally model) we want to learn shares these properties. These properties are just the ones shared between neural networks.
- Every time we choose an ML/DL algo or architecture we are implicitly inducing an inductive bias on our model, a set of prior belief that we assume to be true.
- deep nn good.

6.5 Backpropagation

I skipped most part here, as I think it's significantly more pedagogical to look at illustrations/animations/implementations than reading pseudocode for these things. I also skipped the time complexities of the operations. The bigger point is that automatic differentiation is an easier to understand approach to implementing backpropagation rather than performing analytical approach where you derive all the gradients yourself by hand and then implement. What's interesting is that if you encapsulate the differentiation rules for different types of gradients, then you can create this automatic differentiation framework that abstracts away from manually computing the gradients, and instead focus on just applying the chain rule and multiply them together.

6.6 Historical notes

Some interesting history, read if curious, I didn't pay too much attention to it.

6.7 Summary 6.1-6.3

Section 6.1 showed limitations of linear models thus motivating the need for non-linearity, which could be achieved in neural nets. The XOR function was investigated in this section showing how a neural net could learn such function that required non-linearity to be solved.

Section 6.2 addressed some common output units and showcased how to analyze the effects these units had on the training process. For instance effects of saturations and numerical instability are important factors to consider when choosing the output unit. And having ways to combat saturation, as many popular activation functions have some saturation problem, is crucial, which can be done with for instance log. The form of cross-entropy is decided by the choice of model, which in turn is decided by the form of output and the nature of the problem (regression or classification). For binary outputs sigmoid with Bernoulli and cost function through cross-entropy can mitigate some of the saturation shortcomings of sigmoid. Softmax for categorical outputs (more than 2) is a popular choice with a clever math trick to make it more numerical stable. Cross-entropy also mitigates some of the saturation issues it has, but not all, so good to be aware.

Section 6.3 dealt with popular hidden units (activation functions) such as relu and tanh. Relu is one of the most common and popular first choice. Choosing activation function consists of trial and error albeit there are practical guidelines, but there's no rigorous underlying theorem for it, you don't know until you try. Non-differentiability in activation functions in practise is not an issue, as there are usually only a few points where these occur, and the fact that a model rarely reaches all the way to a local minima makes it OK, so even if there are some bumps along the way, as long as the training can get the model reasonably close to a minima then it should be fine.

Finally, the biggest take-away from these sections isn't necessarily so much about the specific popular activation functions or output units presented, which is useful knowledge but also something you can always look up, but rather what kind of analysis goes into working with these things. You have to keep an eye out for saturation and numerical instabilities and when it can happen, and how to mitigate it in the cost function.

6.8 Summary 6.4-6.6

– No summary –

6.9 What did we learn

- Adding linear layers on top of each other is not entirely useless, it can serve to reduce the number of parameters needed for the weight matrix. The example given in the book shows that if you had two weight matrices, one for each linear layer, then the number of parameters would be $n \cdot q + q \cdot p = (n + p)q$. Compare this to the flattened linear layer where the weight matrix has parameters np. This means that if q is small then there will be savings in the number of parameters, although at the cost of constraining the matrix to be lower rank.
- There's no rigorous theory underlying the choice of activation, it's very much an empirical effort that must be done with trial and error. There are guidelines, and recommended activations to start with such as relu, gelu, silu, swish etc, but you never truly know until you try it.

- Non-linearity in neural networks causes non-convexity for most cost functions thus creating the need for gradient descent.
- Gradient descent sensitive to parameter initialization.
- Some talks about softplus in section 6.2, and then at the end of 6.3 they discourage the use of softplus in practise. In theory it looks better than relu, pretty similar and is differentiable, but in practise there is a study showing it appears to be worse.
- Sigmoid still has applications in more traditional architectures despite it's shortcomings like in autoencoder, time series models (perhaps), RNN and various probabilistic models.
- Even though some activations are not fully differentiable, it's in practise for the most part OK to still use, because the non-differentiable part is usually only located at one location or very very few locations.
- If we specify a probability distribution for the model, then the cost function is automatically specified as well in the form of a cross-entropy meaning a negative log-likelihood.
- I thought it was useful of the author to showcase how to analyze output units and activation functions, for instance where it can saturate and therefore cause issues with training or when it can get unstable and therefore remedy it by some clever math tricks. It sort of shows you the mentality needed to work with these things.
- Automatic differentiation much better than implementing the backprop analytically, as it's easily extensible, much easier to code and also much more structured.

6.10 Difficulty understanding sections

Section 6.2 was by far the hardest if you really wanted to dig into all the claims the author made and prove them. Aside from statements made, just contemplating some things that was said in that section could throw you into some rabbit holes.

I think section 6.5, which is the most important section of the chapter, isn't as pedagogical as I thought. It takes a long time to digest it properly, because while you need the formality if you explain backpropagation in text to avoid ambiguity, it becomes very tedious to read it. A better way in my opinion is either to implement it or watch illustration of it.

6.11 Quotes

> It is best to think of feedforward networks as function approximation machines that are designed to achieve statistical generalization, occasionally drawing some insights from what we know about the brain, rather than as models of brain function.
>
> — Page 169

> The convergence point of gradient descent **depends on the initial values of the parameters**. In practice, gradient descent would usually not find clean, easily understood, integer-valued solutions like the one we presented here.
>
> — Page 177

> The largest difference between the linear models we have seen so far and neural networks is that the nonlinearity of a neural network causes most interesting loss functions to become non-convex.
>
> — Page 177

Most modern neural networks are trained using maximum likelihood.

— Page 178

Previously, we saw that the equivalence between maximum likelihood estimation with an output distribution and minimization of mean squared error holds for a linear model, but in fact, the equivalence holds regardless of the f(x; θ) used to predict the mean of the Gaussian.

— Page 179

One recurring theme throughout neural network design is that the gradient of the cost function must be large and predictable enough to serve as a good guide for the learning algorithm. Functions that saturate (become very flat) undermine this objective because they make the gradient become very small. In many cases this happens because the activation functions used to produce the output of the hidden units or the output units saturate. **The negative log-likelihood helps to avoid this problem for many models.** Many output units involve an exp function that can saturate when its argument is very negative. The log function in the negative log-likelihood cost function undoes the exp of some output units.

— Page 179

The choice of cost function is tightly coupled with the choice of output unit. Most of the time, we simply use the cross-entropy between the data distribution and the model distribution. The choice of how to represent the output then determines the form of the cross-entropy function.

— Page 181

Like the sigmoid, the softmax activation can saturate. The sigmoid function has a single output that saturates when its input is extremely negative or extremely positive. In the case of the softmax, there are multiple output values. These output values can saturate when the differences between input values become extreme. When the softmax saturates, many cost functions based on the softmax also saturate, unless they are able to invert the saturating activating function.

— Page 186

The principle of maximum likelihood provides a guide for how to design a good cost function for nearly any kind of output layer. In general, if we define a conditional distribution $p(y \mid x; \theta)$, the principle of maximum likelihood suggests we use $-\log p(y \mid x; \theta)$ as our cost function.

— Page 188

Gaussian mixture outputs are particularly effective in generative models of speech (Schuster, 1999) or movements of physical objects (Graves, 2013). The mixture density strategy gives a way for the network to represent multiple output modes and to control the variance of its output, which is crucial for obtaining a high degree of quality in these real-valued domains.

— Page 190

It can be difficult to determine when to use which kind (though rectified linear units are usually an acceptable choice). We describe here some of the basic intuitions motivating each type of hidden units. These intuitions can help decide when to try out each of these units. **It is usually impossible to predict in advance which will work best**. The design process consists of trial and error, intuiting that a kind of hidden unit may work well, and then training a network with that kind of hidden unit and evaluating its performance on a validation set.

— Page 192

Because we do not expect training to actually reach a point where the gradient is 0 , it is acceptable for the minima of the cost function to correspond to points with undefined gradient.

— Page 192

Choosing a deep model encodes a very general belief that the function we want to learn should involve composition of several simpler functions. This can be interpreted from a representation learning point of view as saying that we believe the learning problem consists of discovering a set of underlying factors of variation that can in turn be described in terms of other, simpler underlying factors of variation.

— Page 201