Machine Learning and Pattern Recognition - Week 7 - Introduction to Neural Networks

Antonio León Villares

November 2022

Contents

Based on the online notes here.

1 Neural Networks

1.1 Neural Networks for Feature Engineering

- **What is feature engineering?**
  - the **manual** generation of **features** for our models
  - in our case, this includes certain model choices, such as using **basis functions** to create new features from **raw** data
  - however, we always had to manually decide on the **parameters** of the basis functions
  - for instance, with a sigmoidal basis function:

$$\phi_k(\underline{x}) = \sigma((\underline{v}^{(k)})^T \underline{x} + b^{(k)})$$

    where $\underline{v}^{(k)}, b^{(k)}$ are manually chosen weights and biases

- **How do neural networks differ from the models we have seen until now (i.e linear regression, logistic regression, etc...)**
  - we train a model to **automatically** "learn" the features
  - for example, the parameters of the **sigmoidal** model:

$$f(\underline{x}) = \underline{w}^T \underline{\phi}(\underline{x}) + b$$

$$\phi_k(\underline{x}) = \sigma((\underline{v}^{(k)})^T \underline{x} + b^{(k)})$$

    are:

$$\theta = \{\{\underline{v}^{(k)}, b^k\}_{k=1}^K, \underline{w}, b\}$$

  - until now, we have just **optimised** $\underline{w}, b$
  - a **neural network** fits **all** of θ to the data: it learns to extract features **directly**

1.2 Neural Network Terminology

- **What is a unit?**
  - a "building block" for a neural network
  - it performs a computation to map a set of inputs into an output
  - for example, σ can be thought of as a **logistic unit**:

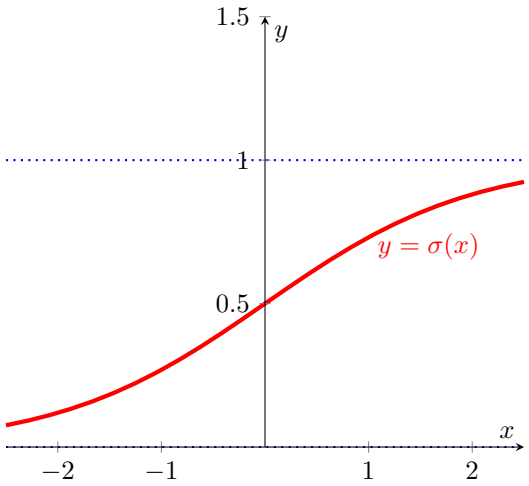
$$\underline{x} \mapsto \sigma(\underline{w}^T \underline{x})$$

- **How do hidden units differ from visible units?**
  - a **hidden unit** is a unit which occur **before** the **output** of the model, and **after** the **input**
  - the **inputs** are known as **visible units**
  - the **hidden units** are what define our model
- **What is a layer of a neural network?**
  - a **collection** of **units**, all of which act on the **same** input, and produce their own output

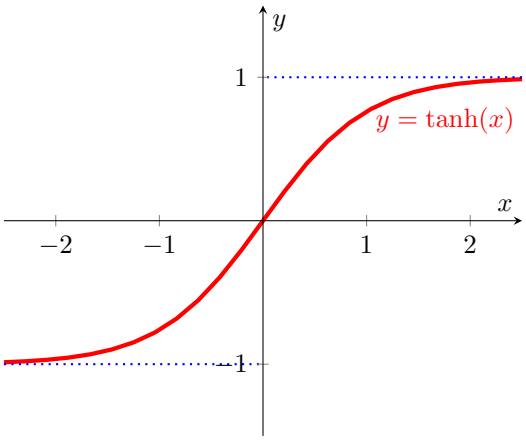
- for example, a set of **basis functions** ϕ_k all act on the **same** input data $\underline{x}$, and produce a different result

- **How do feedforward neural networks compute values at each layer?**
  - a **feedforward** neural network is a type of neural network, where the **outputs** of a layer are **fed forward** as **inputs** of the next layer
  - a **layer** outputs a **vector** by:
    1. applying a **linear transformation** to the input:
    $$\underline{x} \mapsto W^\ell \underline{x} + \underline{b}^\ell$$
    2. applying a **non-linear** function elementwise:
    $$W^\ell \underline{x} + \underline{b}^\ell \mapsto g^\ell(W^\ell \underline{x} + \underline{b}^\ell)$$
  - that is, if we set $\underline{h}^\ell$ to be the output at the ℓth layer, we get that a **feedforward** network is defined by:
  $$\underline{h}^\ell = g^\ell(W^\ell \underline{h}^{\ell-1} + \underline{b}^\ell)$$

- **What types of non-linearities can be used in a neural network?**
  - the **sigmoid**:
  $$\sigma(x) = \frac{1}{1+e^{-x}}$$

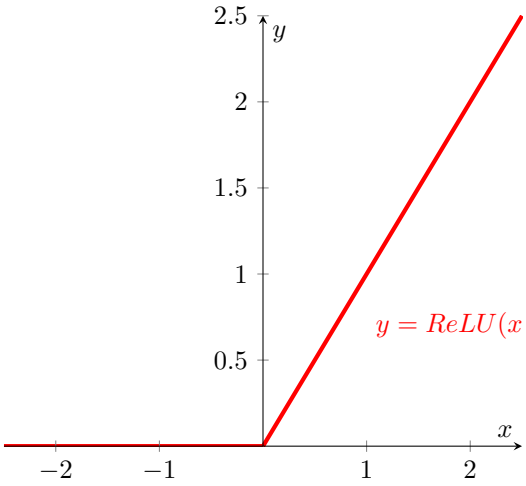


  (in practice this isn't really used)
  - the **hyperbolic tangent**:
  $$\tanh(x)$$



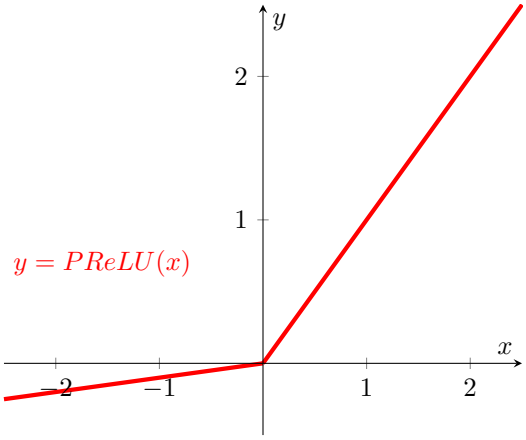
– **ReLU** (rectified linear unit):

$$ReLU(x) = \max(0, x)$$



– **PReLU (parametric ReLU)** or **Leaky ReLU**:

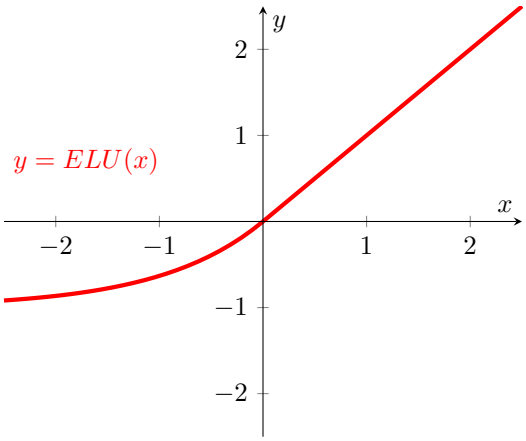
$$PReLU(x) = max(\alpha x, x), \qquad \alpha \in [0, 1]$$



– **ELU (exponential linear unit)**:

$$ELU(x) = \begin{cases} x, & x \geq 0 \\ \alpha(e^x - 1), & x < 0 \end{cases}$$

(also exists **SELU** (scaled ELU))



- **Why are differentiable non-linearities necessary?**
  - to train a neural network, we use **gradient optimisiation**, so differentiability is important

1.3 Neural Networks as Directed Acyclic Graphs

It can be hard to think of units and layers abstractly. Typically, when discussing neural networks, we think of them as **directed acylic graphs**: the **nodes** are the **units**, whilst the **edges** correspond to how layers connect amongst themselves.

For example, if we have a single input $\underline{x} \in \mathbb{R}^3$, and we want to compute:

$$\sigma(\underline{w}^T \underline{x} + b)$$

we can represent this diagrammatically as:

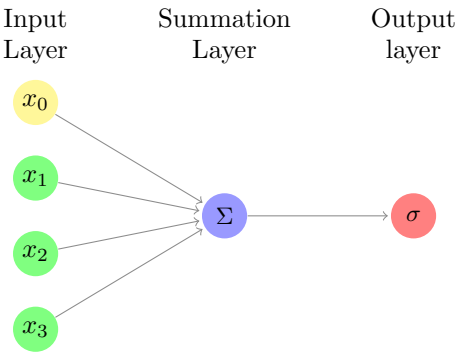
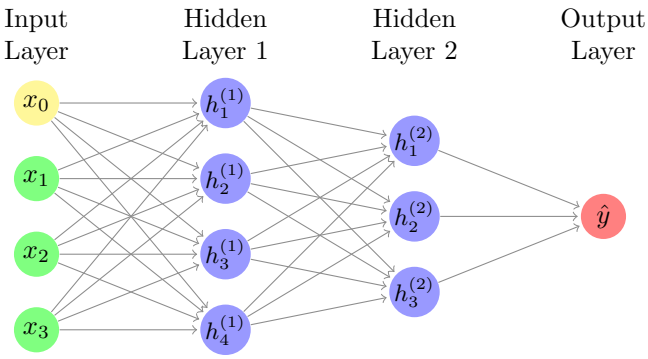


Figure 1: x_0 represents the bias term. We can think of the weights acting within the edges (i.e when x_1 gets added to the summation unit, it has had w_1 applied to it).

If we then choose to add more complexity with hidden layers, our network looks like:



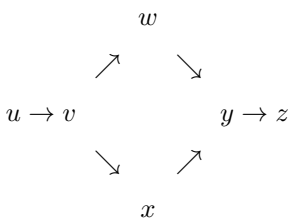
2 Training Neural Networks

2.1 Computing Derivatives in a Neural Network

- **How are neural networks trained?**
  - we used **gradient-based** optimisations
  - for this, it is quite useful to think of neural networks as **DAGs** of function composition
  - for instance, if our network was defined by:

$$z = \exp(\sin(u^2)\log(u^2))$$

we can represent this as a graph:

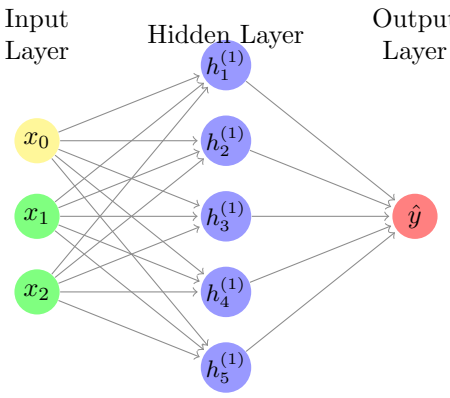


where:

$$\begin{aligned}
v &= u^2 & \frac{\partial v}{\partial u} &= 2u \\
w &= \sin(v) & \frac{\partial w}{\partial v} &= \cos(v) \\
x &= \log(v) & \frac{\partial x}{\partial v} &= \frac{1}{v} \\
y &= wx & \frac{\partial y}{\partial w} &= x \quad \frac{\partial y}{\partial x} = w \\
z &= \exp(y) & \frac{\partial z}{\partial y} &= \exp(y) = z
\end{aligned}$$

- **Is the loss function for a neural network convex?**
  - in general no: we won't have a **convex** function of all the parameters

– to see why, say you have found a **optimum** of the weights (such that any change to them will increase the loss):



– then, if we swap the parameters of $h_1^{(1)}$ with the parameters of $h_2^{(\ell)}$, we are technically just changing the order in which the units appear within the layer, so we will train the exact **same** network (since gradients will swap accordingly)

– hence, the cost of this "swapped" network will be the **same** as the original, and the **optimum** wasn't unique, so the function can't be convex

– that's why **gradient-based** methods are the best option - although they are liable to get stuck in **local optima**

2.1.1 Forward-mode Differentiation

- **What is forward-mode differentiation?**

  – we compute the **gradients** of our variables with respect to the **scalar input**

  – for the example above, if:

$$\dot{\theta} = \frac{\partial \theta}{\partial u}$$

  then we have:

Variable	Derivative wrt Input	Forward-Mode Derivative
u	$\frac{\partial u}{\partial u} = 1$	$\dot{u} = \frac{\partial u}{\partial u} = 1$
$v = u^2$	$\frac{\partial v}{\partial u} = 2u$	$\dot{v} = \frac{\partial v}{\partial u} = 2u$
$w = \sin(v)$	$\frac{\partial w}{\partial v} = \cos(v)$	$\dot{w} = \frac{\partial w}{\partial u} = \frac{\partial w}{\partial v}\frac{\partial v}{\partial u} = \cos(v)\dot{v}$
$x = \log(v)$	$\frac{\partial x}{\partial v} = \frac{1}{v}$	$\dot{x} = \frac{\partial x}{\partial u} = \frac{\partial x}{\partial v}\frac{\partial v}{\partial u} = \frac{1}{v}\dot{v}$
$y = wx$	$\frac{\partial y}{\partial w} = x,\ \frac{\partial y}{\partial x} = w$	$\dot{y} = \frac{\partial y}{\partial u} = \frac{\partial y}{\partial w}\frac{\partial w}{\partial u} + \frac{\partial y}{\partial x}\frac{\partial x}{\partial u} = x\dot{w} + w\dot{x}$
$z = \exp(y)$	$\frac{\partial z}{\partial y} = \exp(y) = z$	$\dot{z} = \frac{\partial z}{\partial u} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial u} = z\dot{y}$

  – notice, once we compute the actual functions (i.e $x = \log(v), w = \sin(v)$), computing and propagating derivatives will generally be **cheaper**: it just involves basic arithmetic operations

- **What are the benefits of forward-mode differentiation?**

- the **forward derivatives** can be computed **at the same time** (i.e in **parallel**) as the actual neural network computation
- this is because, for instance, to compute $\dot{y}$ we only need knowledge of what happened with w, x and v - knowledge of the value of z won't affect $\dot{y}$
- we just require extra memory to store the derivatives comptued
- these derivatives can be computed **automatically** by using forward-mode automatic differentiation

2.1.2 Backward-mode Differentiation

- **What is backward-mode differentiation?**
  - we compute the **gradients** of our **output** with respect to the **variables**
  - for the example above, if:

$$\bar{\theta} = \frac{\partial z}{\partial \theta}$$

then we have:

Variable	Derivative wrt Input	Backward-Mode Derivative
u	$\frac{\partial u}{\partial u} = 1$	$\bar{u} = \frac{\partial z}{\partial u} = \frac{\partial z}{\partial v}\frac{\partial v}{\partial u} = \bar{v}(2u)$
$v = u^2$	$\frac{\partial v}{\partial u} = 2u$	$\bar{v} = \frac{\partial z}{\partial v} = \frac{\partial z}{\partial w}\frac{\partial w}{\partial v} + \frac{\partial z}{\partial x}\frac{\partial x}{\partial v} = \bar{w}\cos(v) + \bar{x}\frac{1}{v}$
$w = \sin(v)$	$\frac{\partial w}{\partial v} = \cos(v)$	$\bar{w} = \frac{\partial z}{\partial w} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial w} = \bar{y}x$
$x = \log(v)$	$\frac{\partial x}{\partial v} = \frac{1}{v}$	$\bar{x} = \frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x} = \bar{y}w$
$y = wx$	$\frac{\partial y}{\partial w} = x,\ \frac{\partial y}{\partial x} = w$	$\bar{y} = \frac{\partial z}{\partial y} = z$
$z = \exp(y)$	$\frac{\partial z}{\partial y} = \exp(y) = z$	$\bar{z} = \frac{\partial z}{\partial z} = 1$

  - again, computing/propagating these derivatives is relatively cheap
- **What are the drawbacks of backward-mode differentiation?**
  - unlike with **forward differentiation**, we can no longer compute derivatives in **parallel** (to compute $\bar{x}$, we need to know what has happened to y and z, which still haven't been computed)
  - **backpropagation** is also **harder** to implement, and requires **more memory**
- **Why is backpropagation the algorithm of choice for neural netwroks?**
  - we can compute **all** derivatives in **one sweep** (i.e one function evaluation)
  - this includes both derivatives with respect to the **input** ($\dot{\theta}$) and with respect to the **variables** ($\bar{\theta}$)
  - a **forward derivative** scheme would require passing through the network once **for each** derivative $\bar{\theta} = \frac{\partial z}{\partial \theta}$, since we need to compute z, and then differentiate wrt the variable of choice
  - hence, **backpropagation** will be many tiems faster than **forward-mode differentiation**

2.2 Array-Based Derivatives

*In practice, neural networks are defined by **matrix multiplication**, not **scalar multiplication**. We know explore how gradients are implemented when matrix multiplication is at play.*

- **How are derivatives computed in practice?**
  - if derivatives were computed based on **scalar** operations, they would be **slow**
  - instead, specialised libraries (BLAS, LAPACK) have been **optimised** for certain **matrix** operations
- **How does backpropagation look like for matrix operations?**
  - we can consider the following graph, where a matrix C is produced by combining 2 matrices A, B:

$$\begin{array}{ccc} \cdots \to A & & \\ & \searrow & \\ & & C \to \cdots \to z \\ & \nearrow & \\ \cdots \to B & & \end{array}$$

  - using **backpropagation**:

$$\bar{A}_{ij} = \frac{\partial z}{\partial A_{ij}} = \sum_{k,l} \frac{\partial z}{\partial C_{kl}} \frac{\partial C_{kl}}{\partial A_{ij}} = \sum_{k,l} \bar{C}_{kl} \frac{\partial C_{kl}}{\partial A_{ij}}.$$

    where for a matrix X:

$$\bar{X}_{ij} = \frac{\partial z}{\partial X_{ij}}$$

- **How are backpropagation gradients computed in practice?**
  - if we naively computed each term:

$$\bar{A}_{ij} = \frac{\partial z}{\partial A_{ij}} = \sum_{k,l} \frac{\partial z}{\partial C_{kl}} \frac{\partial C_{kl}}{\partial A_{ij}} = \sum_{k,l} \bar{C}_{kl} \frac{\partial C_{kl}}{\partial A_{ij}}.$$

    we'd perform $\mathcal{O}(N^4)$ operations (assuming $A, C \in \mathbb{R}^{N \times N}$)
  - instead, we derive **matrix derivative rules**, such that we can compute $\bar{C}$ with respect to $\bar{A}, \bar{B}$:
    1. **Matrix Product**:
$$C = AB \implies \bar{A} = \bar{C}B^T \text{ and } \bar{B} = A^T\bar{C}$$
    2. **Matrix Addition**:
$$C = A + B \implies \bar{A} = \bar{C} \text{ and } \bar{B} = \bar{C}$$
    3. **Element-Wise Function**:
$$C = g(A) \implies \bar{A} = g'(A) \odot \bar{C}$$
    where $\odot$ is the **element-wise/Hadamard** product
    4. **Masking**:
$$C = r(A) \implies \bar{A} = r(\bar{C})$$
    where r masks out elements of A (i.e returning the lower triangular section of A)
    5. **Reshaping**:
$$C = r(A) \implies \bar{A} = r^{-1}(\bar{C})$$
    where r **rearranges** the elements of A, and r^{-1} **reverses** this operation (i.e $r(A) = A^T$, or $r(A) = A_{*j}$ which extracts the jth column, where the derivative of elements which get removed from the matrix will be 0)

2.2.1 Backpropagation in Feedforward Networks

We begin by fixing notation:

- $^{(\ell)}$ will denote that we are operating with parameters in the ℓth layer of the network (there are L many layers, such that if $\ell = L$, we are at the output layer)
- $K^{(\ell)}$ denotes the number of units in layer ℓ
- $W^{(\ell)}$ is the weight matrix in layer ℓ, where:
$$W^{(\ell)} \in \mathbb{R}^{K^{(\ell)} \times K^{(\ell-1)}}$$
- $\underline{b}^{(\ell)}$ is a vector of biases:
$$\underline{b}^{(\ell)} \in \mathbb{R}^{K^{(\ell)}}$$
- if we work on a minibatch of B samples, the output of layer ℓ before applying a non-linearity is a matrix known as the **activation**:
$$A^{\ell}$$
(here, the training samples are stored as **columns**)
- after applying the non-linearity, we obtain the output of a hidden layer, again a matrix when working with minibatches:
$$H^{\ell} \in \mathbb{R}^{K^{(\ell)} \times B}$$

Using this, we can write the activation of layer ℓ as:

$$A^{(\ell)} = W^{(\ell)} H^{(\ell-1)} + \underline{b}^{(\ell)} \underline{1}_B^T$$

where $\underline{b}^{(\ell)} \underline{1}_B^T \in \mathbb{R}^{K^{(\ell)} \times B}$ is a matrix of biases with the bias vector $\underline{b}^{(\ell)}$ as columns.

The output of the ℓth layer will thus be:

$$H^{(\ell)} = g^{(\ell)}(A^{(\ell)})$$

and the output of the network will be $H^{(L)}$ (or $H^{(L)T}$ is we want the output for each sample to be a row vector)

An equivalent formulation can be done if we operate over a single sample:

$$\underline{a}^{(\ell)} = W^{(\ell)} \underline{h}^{(\ell-1)} + \underline{b}^{(\ell)} \qquad \underline{h}^{(\ell)} = g^{(\ell)}(\underline{a}^{(\ell)})$$

Now, let:

$$c = L(F, Y)$$

denote our **cost** (where F is the output for the network, and Y is a matrix of targets)

We apply backpropagation (with respect to the cost c) by using the aforementioned matrix derivative rules to get:

Variable	Backpropagation Derivative
F	$\bar{F} = \frac{\partial c}{\partial F} = \frac{\partial c}{\partial H^{(L)}}$
$H^{(\ell)} = g^{(\ell)}(A^{(\ell)})$	$\bar{A^{(\ell)}} = \frac{\partial c}{\partial A^{(\ell)}} = g'(A^{(\ell)}) \odot \bar{H}^{(\ell)}$
$A^{(\ell)} = W^{(\ell)} H^{(\ell-1)} + \underline{b}^{(\ell)} \underline{1}_B^T$	$\bar{H}^{(\ell-1)} = W^{(\ell)T} \bar{A}^{(\ell)}$
$W^{(\ell)}$	$\bar{W}^{(\ell)} = \bar{A}^{(\ell)} H^{(\ell-1)T}$
$\underline{b}^{(\ell)}$	$\underline{\bar{b}}^{(\ell)} = \bar{A}^{(\ell)} \underline{1}_B$

3 Network Architectures

3.1 Vector Embeddings

- **What is a vector embedding?**
  - a **dense** vector representation of a non-mathematical object
  - for instance, **words**
- **How can we use neural networks to generate vector embeddings?**
  - assume we have a one-hot representation of data (i.e words, were 1 at position d denotes the dth word in some vocabulary)
  - the first layer of our network is given by:

$$\underline{e} = g(W\underline{x} + \underline{b})$$

  - notice, when our one-hot embedding goes in, it only "interacts" with the dth column of W (since all but the dth row of $\underline{x}$ will be 0):

$$\underline{e} = g(W\underline{x} + \underline{b}) = g(\underline{w}_d^T \underline{x} + b)$$

  - once we have trained our network for some task, we can use $\underline{w}_d$ as our **vector embedding** for the dth word in the vocabulary
  - in practice, an **embedding layer** is used: instead of performing the costly multiplication $W\underline{x}$, we have a lookup table which gives us learnable parameters for each distinct input $\underline{x}$ (these parameters are what becomes the embedding)

3.2 Changing Layer Types

3.2.1 Residual Layers

Check this article to see how skip connections are used for ResNets, and this article on how residual layers are used in ResNets.

- **What is a skip connection?**
  - when the **output** from a layer can be used as input for layers further down the network, not just the next layer

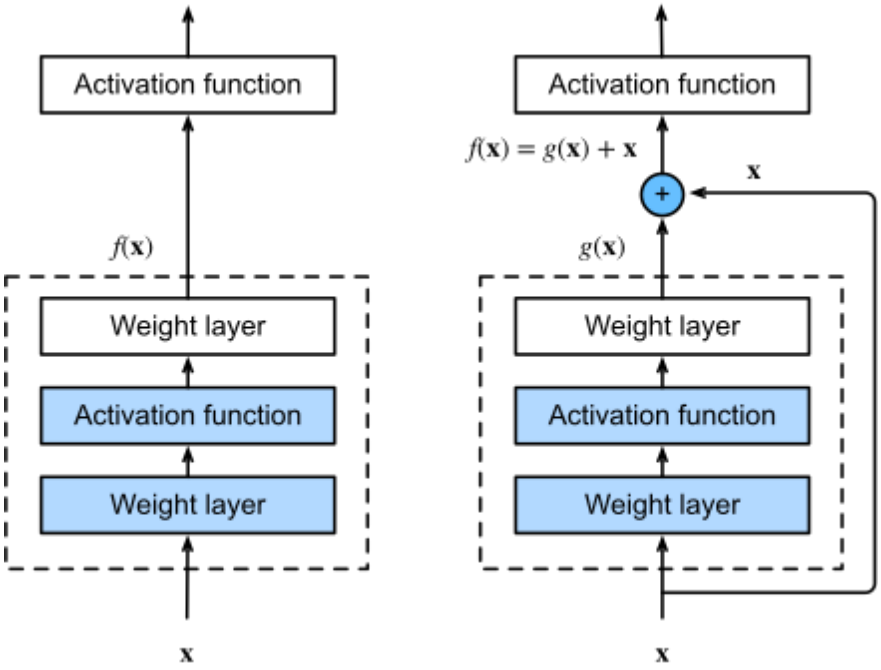
- the idea is to create a **flexible** network, which learns to "turn off" individual hidden layers, based on the input

- **What is a residual layer?**
  - a special case of a **skip connection**
  - we learn weights which allow us to "convert" one hidden layer into another. In particular, the output of a layer, given a previous layer $\underline{h}$ is:

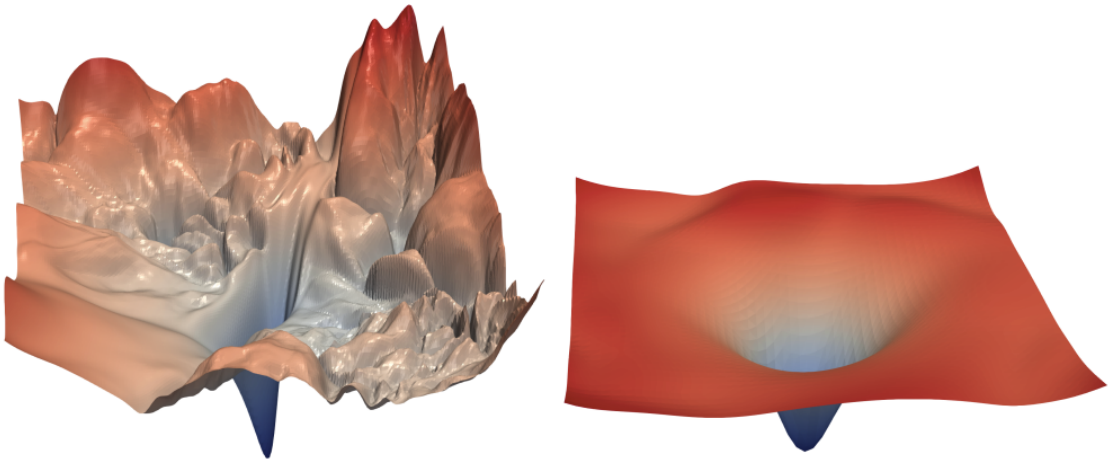
$$r(\underline{h}) = \underline{h} + g(W\underline{h} + \underline{b})$$

  - $g(W\underline{h} + \underline{b})$ is our **residual**, which allows us to recover layer $r(\underline{h})$ from layer h
  - in this way, a **residual layer** allows us to keep "memory" of what happened previously in the network

- **What are the benefits of residual layers?**
  - **Larger Networks**: by using **residual layers**, we can compute gradients without passing through activations. This reduces the risk of **vanishing** and **exploding** gradients, which means that we can train much deeper networks.



  - **Smoother Loss**: residual blocks have been shown to produce loss functions which are **smoother** and also more **convex-like**

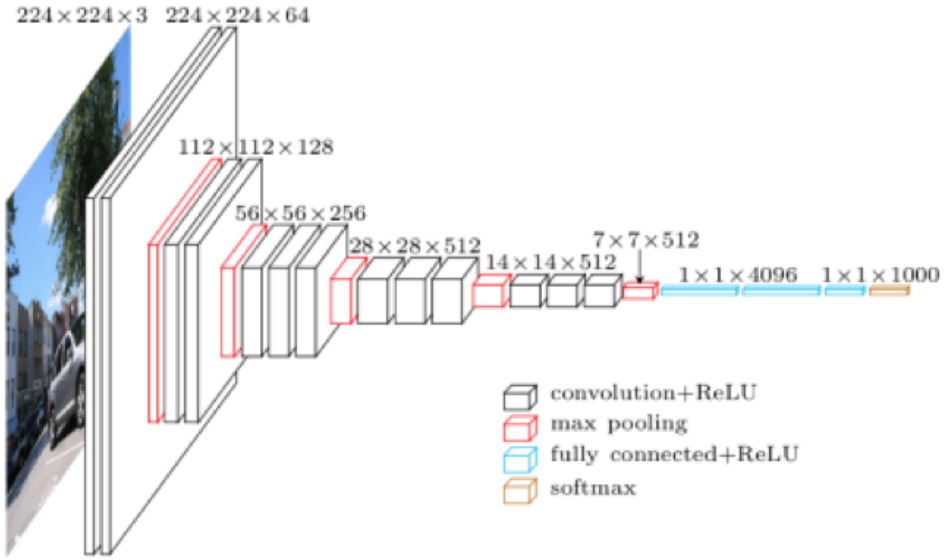


(a) without skip connections (b) with skip connections

Figure 2: Figure by Li et al.

3.3 Convolutional Neural Networks for Images

- **What is a convolutional neural network?**
  - a neural network which takes **images** (i.e 2D or 3D arrays - if there are colour channels) as inputs, and outputs a set of arrays, typically in 3D
  - these arrays correspond to **features** extracted from the image via **convolution kernels**, which act as "filters" of the image to extract interesting features (i.e edges, shapes, etc...)



3.4 Neural Networks for Sequences

3.4.1 Pooling for Sequences of Items

- **Do models always have to have fixed sized inputs?**

- there are many applications in which variable size inputs are desirable
- for example, if we want to **translate** sentences, we can't expect every sentence to have the same length
- if we want to **predict** future behaviour, we would like to be able to do so based on an entire **history**, not just the most recent subset

- **How can variable sized-inputs be included in a model if we know some maximum size?**
  - if we have some **maximum** input size, we can always apply a **padding** to inputs which don't reach that size
  - for instance, if we have a sentnece, we can vectorise it by:
    * concatenating the **word embeddings** of present words
    * using a "dummy" token to represent "missing words"
  - then, we'd always obtain vectors of fixed length
  - however, this is problematic: it can lead to large dimensional input vectors

- **How can pooling be used to represent arbitrary inputs?**
  - instead of **concatenating** inputs, we can use **pooling** to obtain a **representative** for a possible sequence of inputs
  - for example, a **sentence** can be represented as the **average** of its word embeddings:

$$\frac{1}{T}\sum_{t=1}^{T} embedding(\underline{x}^{(t)}; V)$$

    where V are the weights of our embedding model. This works for any set of T inputs.
  - other alternatives include:
    * **max pooling**
    * **sum pooling**
    * **weighted average**
  - **How does a weighted average pooling model work?**
    * we use a **weighted average**:

$$\sum_{t=1}^{T} a(\underline{e}^{(t)})\underline{e}^{(t)}$$

      where:
      · $\underline{e}^{(t)}$ is an **embedding**
      · $a(\underline{e}^{(t)})$ is a scalar weight

      ,
    * we can use **softmax** to compute $a(\underline{e}^{(t)})$

$$\underline{a} = softmax(E\underline{q})$$

      where:
      · E is the matrix with embeddings as rows
      · $\underline{q}$ is a vector of parameters
    * $a(\underline{e}^{(t)})$ is known as **attention**: it allows the model to focus on specific parts of a sentence

3.4.2 RNNs and Transformers

More on the transformer architecture here.

- **What is a RNN?**
  - a **recurrent neural network** is a network architecture for dealing with data sequences
  - the input at a layer ℓ is not onlyt he previous layer, but also part of the sequential input
  - this can be used, for instance, to model **temporal information**
- **What is a transformer?**
  - a special type of RNN, which uses the **attention mechanism**
  - it **transforms** a sequence (i.e words), into another sequence of the same length
  - as such, it has found widespread use for machine translation and other lagnuage based tasks, but also for image generation

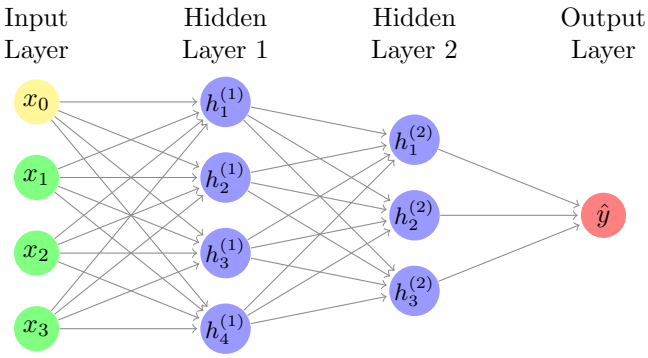
4 Preventing Overfitting in Neural Netwroks

4.1 Initialising Weights

- **Why should weights not be initialised to constants?**
  - say we initialise all hidden units in a layer with the **same** weight and bias parameters:

$$\underline{w}^{(\ell)}, \underline{b}^{(\ell}$$

  - then, our network will look something like:



  but:

$$\forall i, h_i^{(\ell)} = g(\underline{w}^{(\ell)T}\underline{h}^{(\ell-1)} + b^{(\ell)})$$

  - in other words, each **unit** in a **layer** will output the **exact same** value
  - thus, when **backpropagating**, the gradient of each unit in a layer will be the **same**, so all the weights within a layer will be the same
  - as such, our network will depend on a simple linear combination of the inputs
- **Why should weights not be initialised a using a standard normal distribution?**
  - say we choose to initialise weights **randomly**, such that:

$$\underline{w}_i^{(\ell)} \sim \mathcal{N}(\underline{0}, \mathbb{I})$$

- we know make an illustrative assumption, and assume that each input is such that:

$$x_d \in -1, 1, \qquad d \in [1, K^{(\ell)}]$$

- then, the **activation** at a unit will be:

$$(\underline{w}_i^{(\ell)})^T \underline{x}$$

- this is a **random walk**, and as a sum of iid normal variables:

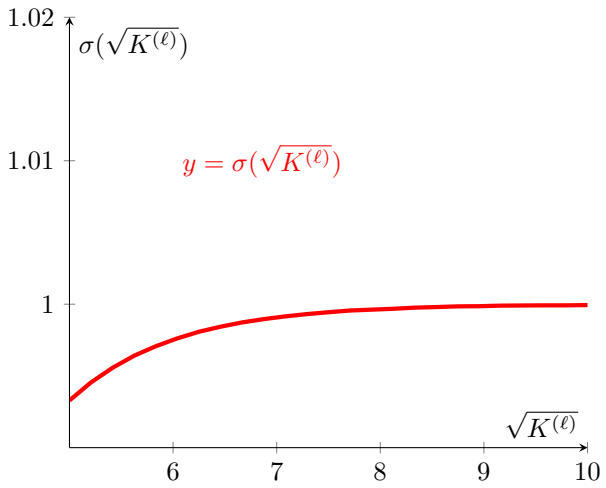
$$\mathbb{E}[(\underline{w}_i^{(\ell)})^T \underline{x}] = \left(\mathbb{E}[(\underline{w}_i^{(\ell)})\right)^T \underline{x} = 0$$

$$Var[(\underline{w}_i^{(\ell)})^T \underline{x}] = \sum_{d=1}^{K^{(\ell)}} Var(w_{id}^{(\ell)}) = K^{(\ell)}$$

- this is problematic depending on the type of activation function which we use
- for instance, if $g = \sigma$, then we expect the average activation to have a magnitude of $\approx \sqrt{K^{(\ell)}}$ (the standard deviation of an activation); but then:

$$g(a_i) \approx \frac{1}{1 + \exp(-\sqrt{K^{(\ell)}})}$$

and if we plot this for $\sqrt{K^{(\ell)}} \geq 5$:



- that is, the sigmoid gets **saturated**: moving away from 0 produces either 1 or 0, since $\sqrt{K^{(\ell)}}$ is large
- hence, as the dimension of our inputs increases (nowadays we can easily use $D \approx 1000$), our sigmoid will get more and more saturated, making it an ineffective activation, since the gradients will be close to 0 (no matter the input, we expect a similar output)

- **How are weights initialised nowadays?**
  - a quick fix for the above problem would be to instead sample using:

$$\mathcal{N}\left(0, \frac{0.01}{K^{(\ell)}}\right)$$

  - a very common initialisation is the **Glorot** (or **Xavier**) initialisation, of which there are 2 types (more on them here):

1. **Normal Glorot**:
$$\mathcal{N}\left(0, \frac{2}{K^{(\ell)} + K^{(\ell-1)}}\right)$$
2. **Uniform Glorot**
$$\mathcal{U}\left(-\sqrt{\frac{6}{K^{(\ell)} + K^{(\ell-1)}}}, \sqrt{\frac{6}{K^{(\ell)} + K^{(\ell-1)}}}\right)$$
where $\mathcal{U}(a, b)$ denotes a **uniform distribution** over the interval $[a, b]$

– otherwise, certain **architectures** have **specialised** initialisations

4.2 Regularisation

- **What is early stopping regularisation?**
  - stopping the training of a neural network **before** it has reached a local optimum
  - typically, we implement by:
    1. Periodically monitor model performance
    2. If validation loss is better than what we have seen thus far, store the weights
    3. If after N checks the validation score hasn't improved, stop optimising and return the saved weights
- **How can we train a neural network with L^2 regularisation?**
  - we can incorporate the term $\lambda\|\underline{\theta}\|^2$ directly into our loss
  - alternatively, we can **noisify** the input:
$$x_i \to x_i + \varepsilon \sim \mathcal{N}(0, 1)$$
which has been shown to have the same effect as standard L^2 regularisation
- **What is dropout regularisation?**
  - we introduce a **dropout layer**, which randomly "turns off" (set to 0) certain hidden units when training the network and backpropagating
  - the idea is to make the hidden units more **robust**: they can no longer depend on all other units, so they must learn weights which fit well to the data (not, for instance, to correct mistakes made by other units)
  - in essence, we are training many different networks at the same time
  - more on dropout (with code) here
- **How does a batch normalisation layer lead to regularisation?**
  - **batch normalisation** is the process of ensuring that the output of a hidden layer is distributed in the same way
  - before outputting a value, the values of a **hidden layer** are normalised by using their mean and standard deviation
  - the idea is that the distribution of inputs across layers will be the same, which improves **generalisation** and **speeds up** convergence to a local optimum (since the layers won't be so dependent on strong signals from previous layers, so they can learn more independently)
  - more can be read:
    * Why Batch Normalization?
    * Towards Data Science
    * Machine Learning Mastery

5 Question

5.1 Notes Questions

1. **We require differentiable non-linearities, but ReLU is not differentiable at $x = 0$. Still, we use ReLU, and for "bad" inputs, we just return a gradient of 0. Can we then use non-differentiable functions, such as the *hard-step function*:**

$$\Theta(a) = \begin{cases} 1, & a > 0 \\ 0, & a \leq 0 \end{cases}$$

   - $\Theta(\underline{w}^T \underline{x})$ will have 0 gradients with respect to the weights $\underline{w}$ (and undefined gradient if $\underline{w}^T \underline{x} = 0$), so the network won't learn
   - with ReLU, gradients will be non-zero for positive inputs, so defining a 0 gradient at the origin won't hinder learning

6 Tutorial

1. **Classification tasks often involve rare outcomes, for example predicting click-through events, fraud detection, and disease screening. We'll restrict ourselves to binary classification, $y \in \{0, 1\}$, where the positive class is rare: $P(y = 1)$ is small.**
   We are likely to see lots of events before observing enough rare events to train a good model. To save resources, it's common to only keep a small fraction f of the $y = 0$ "negative examples". A classifier trained naively on this sub-sampled data would predict positive labels more frequently than they actually occur. For generative classifiers we could set the class probabilities based on the original class counts (or domain knowledge). This question explores what to do for logistic regression.
   We write that an input-output pair occurs in the world with probability $P(\underline{x}, y)$, and that the joint probability of an input-output pair $(\underline{x}, y)$ *and* keeping it (k) is $P(\underline{x}, y, k)$. Because conditional probabilities are proportional to joint probabilities, we can write:

$$P(y \mid \underline{x}, k) \propto P(\underline{x}, y \mid k) \propto P(\underline{x}, y, k) = \begin{cases} P(\underline{x}, y), & y = 1 \\ fP(\underline{x}, y), & y = 0 \end{cases}$$

   (a) **Show that if:**

$$P(y \mid \underline{x}) \propto \begin{cases} c, & y = 1 \\ d, & y = 0 \end{cases}$$

   then:

$$P(y = 1 \mid \underline{x}) = \sigma(\log c - \log d)$$

   As with the work last week, we can write:

$$P(y = 1 \mid \underline{x}) = \frac{c}{c + d} = \frac{1}{1 + \exp(-(\log c - \log d)}$$

   (b) **We train a logistic regression classifier to match subsampled data, so that:**

$$P(y = 1 \mid \underline{x}, k) \approx \sigma(\underline{w}^T \underline{x} + b)$$

   Use the above results to argue that we should add $\log f$ to the bias parameter to get a model for the real-world distribution $P(y \mid \underline{x}) \propto P(y, \underline{x})$.

We write:

$$P(y=1 \mid \underline{x}, k) = \frac{P(y=1 \mid \underline{x})}{P(y=1 \mid \underline{x}) + fP(y=0 \mid \underline{x})}$$

so using the work above:

$$P(y=1 \mid \underline{x}, k) = \sigma\left(\log P(y=1 \mid \underline{x}) - P(y=0 \mid \underline{x}) - \log f\right)$$

Matching the arguments of the sigmoids:

$$\log P(y=1 \mid \underline{x}) - P(y=0 \mid \underline{x}) - \log f = \underline{w}^T \underline{x} + b$$

so:

$$\log P(y=1 \mid \underline{x}) - P(y=0 \mid \underline{x}) = \underline{w}^T \underline{x} + (b + \log f)$$

Hence, if we want to represent $P(y \mid \underline{x})$:

$$P(y=1 \mid \underline{x}) = \sigma(\log P(y=1 \mid \underline{x}) - P(y=0 \mid \underline{x})) = \sigma(\underline{w}^T \underline{x} + (b + \log f))$$

as required.

This makes intuitive sense: since we have a small number of $y = 1$ samples, adding $\log f$ as a bias (which will be negative), will make the activation more negative. This means that the probability of classifying an instance as class 1 will decrease - the model is less confident in assigning $y = 1$ as a label.

If we had set $f = 1$, we would get back our standard sigmoid, which makes sense.

(c) **We now consider a different approach. We may wish to minimise the loss:**

$$L = -\mathbb{E}_{P(\underline{x},y)}[\log P(y \mid \underline{x})]\,.$$

Multiplying the integrand by:

$$1 = \frac{P(\underline{x},\, \underline{y} \mid k)}{P(\underline{x},\, \underline{y} \mid k)}$$

changes nothing, so:

$$\begin{aligned} L &= -\int \sum_y P(\underline{x}, y \mid k)\, \frac{P(\underline{x}, y)}{P(\underline{x}, y \mid k)}\, \log P(y \mid \underline{x})\, \mathrm{d}\underline{x} \\ &= -\mathbb{E}_{P(\underline{x}, y \mid k)}\left[\frac{P(\underline{x}, y)}{P(\underline{x}, y \mid k)}\, \log P(y \mid \underline{x})\right] \\ &\approx -\frac{1}{N} \sum_{n=1}^{N} \frac{P(\underline{x}^{(n)}, y^{(n)})}{P(\underline{x}^{(n)}, y^{(n)} \mid k)}\, \log P(y^{(n)} \mid \underline{x}^{(n)}) \end{aligned}$$

where $\{\underline{x}^{(n)}, y^{(n)}\}$ come from the subsampled data. This manipulation is a special case of a trick known as importance sampling, which we will see later in lectures in another context. We have converted an expectation under the original data distribution, into an expectation under the subsampling distribution. We then replaced the formal

expectation with an average over subsampled data.
Use the above idea to justify multiplying the gradients for $y = 0$ examples by $\frac{1}{f}$, when training a logistic regression classifier on subsampled data.

Recall, $P(\underline{x}, y \mid k)$ is defined up to a constant:

$$P(\underline{x}, y \mid k) \propto \begin{cases} P(\underline{x}, y), & y = 1 \\ fP(\underline{x}, y), & y = 0 \end{cases}$$

Hence:

$$\frac{P(\underline{x}, y)}{P(\underline{x}, y \mid k)} \propto \begin{cases} 1, & y = 1 \\ \frac{1}{f}, & y = 0 \end{cases}$$

If we have a sample with $y = 1$, the loss becomes:

$$L \propto -\frac{1}{N} \sum_{n=1}^{N} \log P(y^{(n)} = 1 \mid \underline{x}^{(n)})$$

which is the standard log-likelihood loss. On the other hand, with $y = 0$:

$$L \propto -\frac{1}{N} \sum_{n=1}^{N} \frac{1}{f} \log P(y^{(n)} = 0 \mid \underline{x}^{(n)})$$

that is, the gradients for negative samples will get scaled by a factor of $\frac{1}{f}$. This has the effect of upweighting (since $f \in [0, 1]$) the effect of observing negative samples (i.e we need toa ccount for the fact that we are only keeping a proportion f of the negative samples)

(d) **Compare the 2 methods that we have considered for training models base don subsampled data, giving some pros and cons of each.**

- the bias method (adding $\log f$) is easier to train (standard training and just add additional bias at the end); the importance sampling method might be harder to train, since gradients can be multiplied by a large number $1/f$, so we'd need to modify step sizes to ensure we converge well

In the limit of infinite data, the empirical estimate of the average loss in method c) will become accurate, and so we will minimize the same loss as without subsampling. Method b) won't usually minimize that same loss: although b) tries to match the same $P(y \mid \underline{x})$ distribution as c), the inputs are still weighted in the average loss by the modified distribution $P(\underline{x} \mid k), whereasinc)theinputlocationsareweightedbyP(\underline{x})$.

Thus method b) is increasing the importance of getting accurate probabilities in regions where positive examples occur. The expected log probability at test time could be worse for b) as a result, but other error measures such as "precision" and "recall" (non-examinable, but worth looking up) may be better than c).

Resampling a dataset to make it more balanced often makes classifiers work better (under some measures), regardless of whether you need to save storage. One could argue we're implicitly changing the loss function, which we should do directly – but pragmatically, it's often easier to process data and use the software we have.

2. **In lectures we turned a representation of a function into a probabilistic model of real-valued outputs by modelling the residuals as Gaussian noise:**

$$P(y \mid \underline{x}, \theta) = \mathcal{N}(y; f(\underline{x}, \theta), \sigma^2)$$

The noise variance σ^2 is often assumed to be a constant, but it could be a function of the input location $\underline{x}$.
A flexible model could set the variance using a neural network:

$$\sigma(\underline{x})^2 = \exp(\underline{w}^{(\sigma)T} \underline{h}(\underline{x}; \theta) + b^{(\sigma)}$$

where $\underline{h}$ is a vector of hidden unit values. These could be hidden units from the neural network used to compute the function $f(\underline{x}; \theta)$, or there could be a separate network to model the variances.

(a) **Assume that $\underline{h}$ is the final layer of the same neural network used to compute f. How could we modify the training procedure for a neural network that fits f by least squares, to fit this new model?**

Since we now want to fit some probabilistic model, we should use maximum likelihood (i.e minimise negative log-likelihood). We can do this by computing gradients.

Writing the variance as $v = \sigma^2$:

$$c = -\log \mathcal{N}(y;\, f,\, v) = \frac{1}{2} \log v + \frac{1}{2v}(y - f)^2 + \frac{1}{2} \log(2\pi).$$

The derivative of the cost with respect to the mean and variance is:

$$\bar{f} = \frac{\partial c}{\partial f} = \frac{1}{v}(f - y),$$
$$\bar{v} = \frac{\partial c}{\partial v} = \frac{1}{2v} - \frac{1}{2v^2}(y - f)^2$$

we will use these for backpropagation.

We can now compute the backpropagation steps themselves. Say that:

$$f = \underline{w}^{(f)T}\underline{h}^{(f)} + b^{(f)}$$
$$a^{(\sigma)} = \underline{w}^{(\sigma)T}\underline{h}^{(\sigma)} + b^{(\sigma)}$$
$$v = \exp(a^{(\sigma)}),$$

where $\underline{h}^{(f)}$ is the final hidden layer before computing the function, which the last squares neural network will also have. $\underline{h}^{(\sigma)}$ is a hidden layer used to compute the new variance prediction.

Backpropagating variance one step:

$$\bar{a}^{(\sigma)} = \bar{v}\exp(a^{(\sigma)}) = \bar{v}v.$$

Standard results for backpropagation apply to a generic scalar NN activation as follows:

$$a = \underline{w}^T\underline{h} + b \quad \Rightarrow \quad \bar{\underline{w}} = \bar{a}\underline{h}, \quad \bar{\underline{h}} = \bar{a}\underline{w}, \quad \bar{b} = \bar{a}.$$

These apply to the variance activation:

$$\bar{\underline{w}}^{(\sigma)} = \bar{a}^{(\sigma)}\underline{h}^{(\sigma)}, \quad \bar{\underline{h}}^{(\sigma)} = \bar{a}^{(\sigma)}\underline{w}^{(\sigma)}, \quad \bar{b}^{(\sigma)} = \bar{a}^{(\sigma)}$$

and the function value f:

$$\bar{\underline{w}}^{(f)} = \bar{f}\underline{h}^{(f)}, \quad \bar{\underline{h}}^{(f)} = \bar{f}\underline{w}^{(f)}, \quad \bar{b}^{(f)} = \bar{f}.$$

These are all teh gradients required to compute the function with the varaince at the final stage of the network.

If we assume that the hidden layers for variance and function are shared, then:

$$\bar{\underline{h}}^{(L)} = \bar{\underline{h}}^{(\sigma)} + \bar{\underline{h}}^{(f)},$$

and then we backpropagate through the least squares network. In general, fitting the variance network should be quite hard, particularly finding good learning rates.

An alternative could be to compute the log of the square residuals on the training set, and then fit a neural network to fit to these log residuals.

(b) **In the suggestion above, the activation:**

$$a^{(\sigma)} = \underline{w}^{(\sigma)T}\underline{h} + b^{(\sigma)}$$

sets the log of the variance of the observations.

i. **Why not set the variance directly to this activation value, $\sigma^2 = a^{(\sigma)}$?**
- the variance should be positive
- however the NN is unconstrained, so it could fit negative variances

ii. **Why not set the variance to the square of the activation, $\sigma^2 = (a^{(\sigma)})^2$?**
- the square is not monotone: smaller activations won't lead to smaller variances (namely, a and $-a$ would both have the same variance)
- the main problem is that with close to 0 activations, the variance will become nearly 0
- these could lead to extreme gradients, which will lead to unstable fitting procedures
- by taking the log, 0 variances only arise for negative infinity activations

(c) **Given a test input $\underline{x}^{(*)}$, the model above outputs both a guess of an output $f(\underline{x}^{(*)}$ and an error bar $\sigma(\underline{x}^{(*)}$, which indicates how wrong the guess could be. The Bayesian linear regression and Gaussian process models covered in lectures also give error bars on their predictions. What are the pros and cons of the neural network approach in this question? Would you use this neural network to help guide which experiments to run?**

1. **Flexibility**:
   - the NN approach is **heteroscedastic**: the variance depends on the test point
   - Bayesian approach (BLR + GP) use a fixed noise assumption, which won't always be a correct assumption
2. **Fitting Cost**:
   - with N datapoints, and large N, neural networks are cheaper to fit
   - a NN with H hidden units and D dimensional inputs costs $\mathcal{O}(NDH)$; a GP would cost $\mathcal{O}(N^3)$ (due to matrix inversions); BLR is also cubic in the number of basis functions, which are also expensive to fit over high dimensional spaces
   - for smaller datasets, BLR + GP might work well though
3. **Uncertainties**:
   - with Bayesian approaches, uncertainty depends on the observation process: for inputs away from observation, the uncertainty will grow (close to the standard deviation of the prior)
   - with NNs, the network is capable of extrapolating variance to unseen regions

3. **Consider the following classification problem. There are 2 real-valued features x_1 and x_2, and a binary class label. The class label is determined by:**

$$y = \begin{cases} 1 & \textbf{if } x_2 \geq |x_1| \\ 0 & \textbf{otherwise}. \end{cases}$$

(a) **Can this function be perfectly represented by logistic regression, or a feedforward neural network without a hidden layer? Why or why not?**
- no: logistic regression learns a linear decision boundary
- however, this problem has a non-linear decision boundary
- for example, $y = 1$ with $x_2 = 2$ and $x_1 = 1$ or $x_1 = -1$

(b) **Consider a simple classification problem:**

$$y = \begin{cases} 1 & \textbf{if } x_2 \geq x_1 \\ 0 & \textbf{otherwise}. \end{cases}$$

Pick weights such that the neuron:

$$h_1 = \Theta(W_{11}^{(1)} x_1 + W_{12}^{(1)} x_2 + b_1^{(1)}),$$

solves this classification problem. Not that Θ is the hard threshold function:

$$\Theta(a) = \begin{cases} 1 & \textbf{if } a \geq 0 \\ 0 & \textbf{otherwise}. \end{cases}$$

- if $x_2 \geq x_1$, then $x_2 - x_1 \geq 0$
- hence, we can just set:

$$W_{11}^{(1)} = -1 \quad W_{12}^{(1)} = 1 \quad b_1^{(1)} = 0$$

  and the step function takes care of the rest

(c) **Going back to the original classification problem, design a 2 layer feedforward network that represents a function for classification. Use the hard threshold activation function.**

We can build on the work above, by defining a second neuron:

$$h_2 = \Theta(W_{21}^{(1)} x_1 + W_{22}^{(1)} x_2 + b_2^{(1)}),$$

with:

$$W_{21}^{(1)} = 1 \quad W_{22}^{(1)} = 1 \quad b_2^{(1)} = 0$$

This will handle the case when $x_1 < 0$, such that:

$$h_2 = 1 \iff x_1 + x_2 \geq 0 \iff x_2 \geq -x_1$$

Hence, h_1 and h_2 will output 1 **if and only if** $x_2 \geq |x_1|$.

The output layer can just be an `and` gate:

$$f(h_1, h_2) = \Theta(w_1^{(2)} h_1 + w_2^{(2)} h_2 + b^{(2)})$$

where:

$$w_1^{(2)} = 0.5 \qquad w_2^{(2)} = 0.5 \qquad b^{(2)} = -0.6$$