Neural Networks

Mingfei Sun

Foundations of Machine Learning
The University of Manchester

Neural Networks
Mingfei Sun
Foundations of Machine Learning
The University of Manchester

Outline

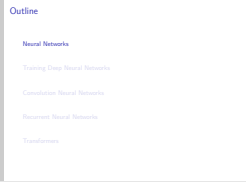
Neural Networks

Training Deep Neural Networks

Convolution Neural Networks

Recurrent Neural Networks

Transformers

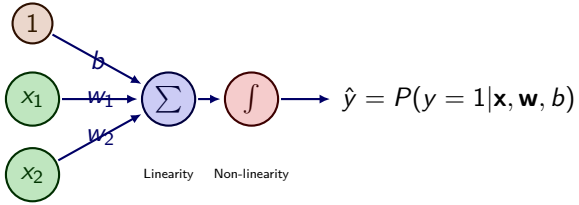


A perceptron takes as inputs $x_1, ..., x_D$, and then outputs sign$(b + w_1x_1 + ... + w_Dx_D)$, where $w_1, ..., w_D$ are the weight parameters of the perceptron, and b is the bias term. We can consider more general models of this kind, which are referred to as artificial neurons. Let $f : \mathbb{R} \to \mathbb{R}$ be an activation function, $x_1, ..., x_D$ the inputs, and $w_1, ..., w_D$ the weights, and b the bias. Then an artificial neuron, also known as a unit, with activation function f outputs the following:

$$f(b + w_1x_1 + w_2x_2 + ... + w_Dx_D).$$

In principle, composing artificial neurons is a powerful idea—essentially an artificial neural network can compute any function that a computer can! Of course, one may then wonder why not try to use models that use boolean circuits directly? A neural network composed of units with continuous activation functions is differentiable end-to-end, i.e., a suitably chosen loss function for any fixed input-output pair is a differentiable function of the weight and bias parameters. Thus, in principle, it may be easier to train such networks than boolean ones. Let us look at a few examples of neural networks.

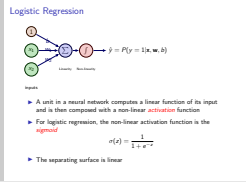
Logistic Regression



- A unit in a neural network computes a linear function of its input and is then composed with a non-linear *activation* function
- For logistic regression, the non-linear activation function is the *sigmoid*

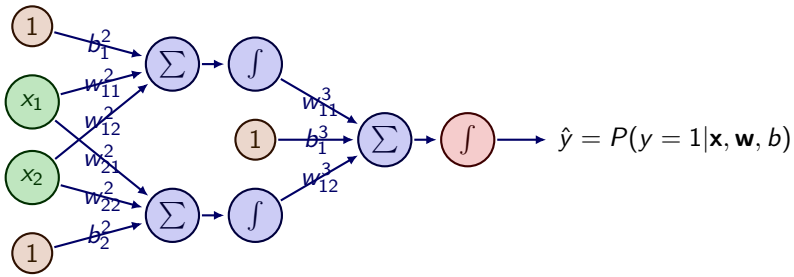
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

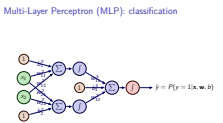
- The separating surface is linear



We can view the logistic regression model as a neural network with a single artificial neuron or unit. The activation function used is the sigmoid function, $\sigma(z) = 1/(1 + e^{-z})$. This is shown schematically in the left figure. The unit is shown as explicitly composed of a "linear function", $b + w_1 x_1 + ... + w_D x_D$ followed by a non-linear activation function, σ. The output of the model is interpreted as the probability that the observed label is 1. As we've already seen, the logistic regression model for classification results in a linear separating surface. Thus, such a simple neural network cannot represent functions of any greater complexity than other models we've seen so far. In order to exploit the full power of neural networks, we need to have larger and deeper "circuits".

Multi-Layer Perceptron (MLP): classification

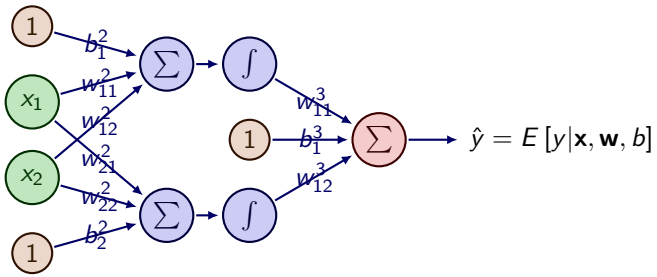


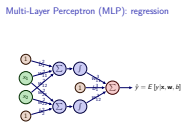


Neural networks composed of more than one "layer" are called multilayer perceptrons (MLPs). Despite the name, the units in the network do not have to be perceptrons, but can in fact be any kind of artificial neurons.

Notation: The weight w_{ij}^l is the weight corresponding to the connection that goes from the j-th unit in the $(l-1)$-th layer to the i-th unit in the l-th layer. For example, w_{21}^2 is the weight on the edge connecting the input x_1 to the second unit in the hidden (second) layer. Succinctly, we can represent this by a matrix $\mathbf{W}^l$ of size $n_l \times n_{l-1}$, where n_l denotes the number of units in the l-th layer. Similarly, we'll denote the bias term of the ith unit in the l-th layer by b_i^l. Succinctly the bias terms for an entire layer are denoted by the vector $\mathbf{b}^l$.

Multi-Layer Perceptron (MLP): regression

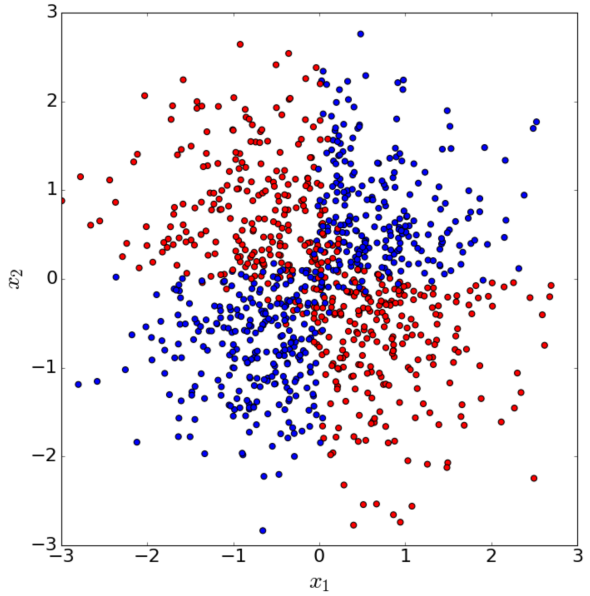


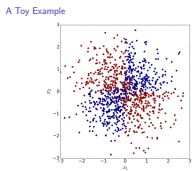


Let us consider a simple multilayer perceptron. There is a lot of notation to unpack in the figure, so let us first go over that. There are *three* layers in the model, the first is referred to as the *input* layer, which just consists of the inputs x_1, x_2. (There are 1s shown as part of input, but essentially they are just accounting for the bias term; in the future we'll drop the 1s and assume that every unit also has a bias term.) The second layer is a *hidden* layer which consists of two hidden units. The layer is said to be hidden because this is not observed as part of the data. Only the inputs and the outputs are observed. In general, it is possible to have more than one hidden layer; indeed, having a large number of hidden layers seems crucial for the success of neural networks on challenging tasks. The third, and in this case final, layer is the *output* layer. As we're solving a classification problem, we'll have a sigmoid activation on the unit in the output layer, so that the output can be interpreted as the probability that the label is 1 (say blue in this case).

By modifying the output layer, we can turn the MLP into a regression model. In this case, the output layer is a linear combination of the outputs from the hidden layer.

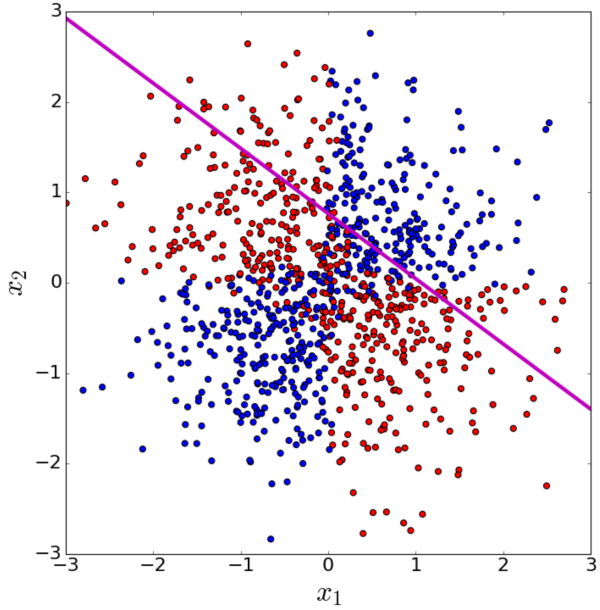
A Toy Example

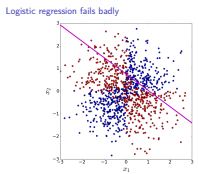




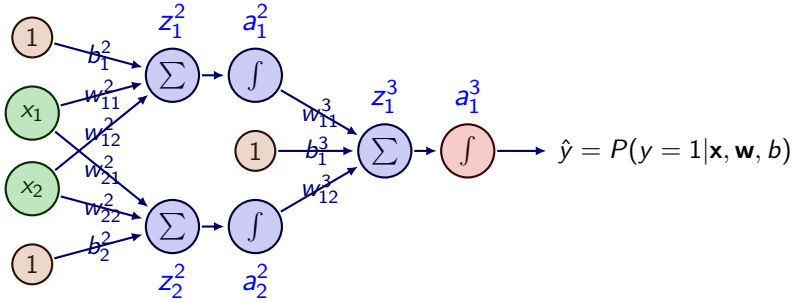
Let us begin by considering a toy example to understand how multilayer perceptrons behave. Let us consider a classification problem, where the data is as shown in the left figure. Clearly, no linear separator can separate the blue points from the red. If we try to train a logistic regression model directly, it fails quite badly, achieving an accuracy not much greater than 50%.

Logistic regression fails badly



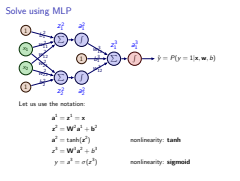


Solve using MLP



Let us use the notation:

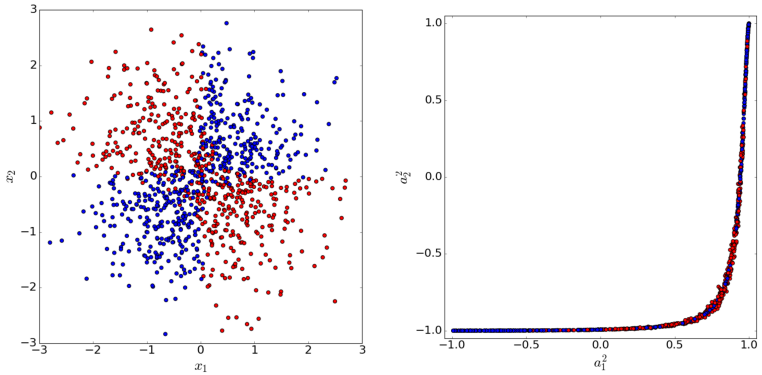
$$\mathbf{a}^1 = \mathbf{z}^1 = \mathbf{x}$$
$$\mathbf{z}^2 = \mathbf{W}^2\mathbf{a}^1 + \mathbf{b}^2$$
$$\mathbf{a}^2 = \tanh(\mathbf{z}^2) \qquad \text{nonlinearity: } \textbf{tanh}$$
$$z^3 = \mathbf{W}^3\mathbf{a}^2 + b^3$$
$$y = a^3 = \sigma(z^3) \qquad \text{nonlinearity: } \textbf{sigmoid}$$

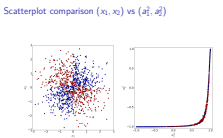


Now we are solving this toy example using MLP.

We first introduce more notations. Typically, every unit in MLP is a linear function followed by a non-linear activation function (if there is no activation function, we'll assume that the activation function is the identity function). Thus, every unit will have a pre-activation value and an activated output, sometimes simply referred to as activation. For unit i in layer l, we'll use z_i^l to denote its pre-activation, i.e., $z_i^l = b_i^l + w_{i1}^l a_1^{l-1} + ... + w_{in_{l-1}}^l a_{n_{l-1}}^{l-1}$, and a_i^l to denote the activation. We'll denote by the vector $\mathbf{z}^l$ and $\mathbf{a}^l$ the preactivations and activations of all the units in layer l. In most cases, the activation $a_i^l = f(z_i^l)$, where f is the activation function of the unit. However, some activation functions apply to an entire layer, notably the *softmax* function. Thus, we'll think of activations as a function directly operating on vectors, $\mathbf{a}^l = f_l(\mathbf{z}^l)$, where f_l is the activation function applied to the entire layer l.

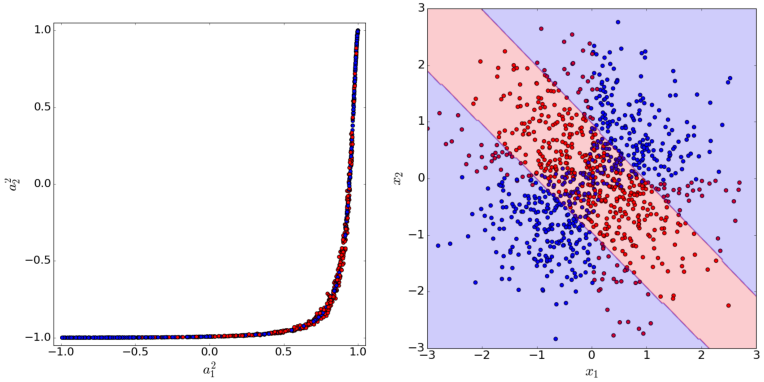
Scatterplot comparison (x_1, x_2) vs (a_1^2, a_2^2)



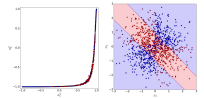


Let us now look at how the MLP model classifies a datapoint as being red or blue. The inputs x_1 , x_2 are transformed into the activations a_1^2, a_2^2 by the hidden layer. These are non-linear and non-local transformations, in that, each of a_1^2 and a_2^2 depends on both x_1 and x_2 and they are non-linear functions of the input. The non-linearity arises due to composition with the hyperbolic tangent function. The final output is simply a logistic regression model, but on the inputs a_1^2, a_2^2 rather than on x_1, x_2. The rightmost figure in the left shows a scatter plot of $(a_1^2, a_2^2$. Although the data is still not entirely linearly separable, it is much more linearly separable in terms of a_1^2 and a_2^2 than it is in terms of x_1 and x_2. Thus, we can view the hidden layer of the MLP as performing "basis expansion" (or rather transformation in this case, as we have not increased the number of features), however, rather than us designing non-linear features, we allow the features themselves to be "learned" as part of training the neural network.

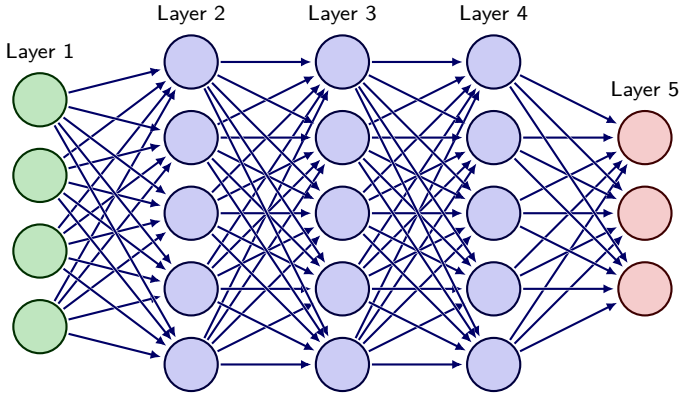
Decision boundary of the neural net

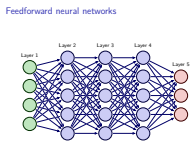


Decision boundary of the neural net



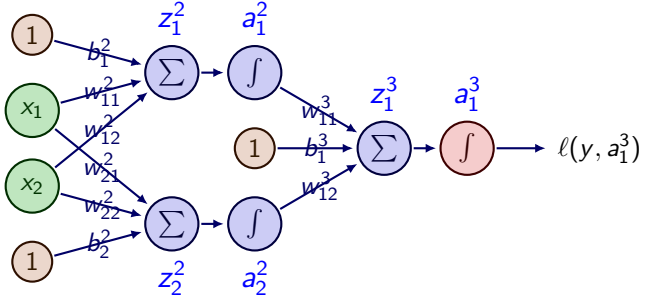
Feedforward neural networks





Feedforward Neural Networks are just a stack of multiple MLPs. There can be multiple hidden layers. Likewise, these hidden layers will transform the input data into non-linear features which may be more effective for downstream tasks. Note that we always count the layer number from the input layer till the output layer.

Compute gradients on a toy example



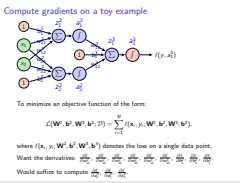
To minimize an objective function of the form:

$$\mathcal{L}(\mathbf{W}^2, \mathbf{b}^2, \mathbf{W}^3, \mathbf{b}^3; \mathcal{D}) = \sum_{i=1}^{N} \ell(\mathbf{x}_i, y_i; \mathbf{W}^2, \mathbf{b}^2, \mathbf{W}^3, \mathbf{b}^3),$$

where $\ell(\mathbf{x}_i, y_i; \mathbf{W}^2, \mathbf{b}^2, \mathbf{W}^3, \mathbf{b}^3)$ denotes the loss on a single data point,

Want the derivatives: $\frac{\partial \ell}{\partial w_{11}^2}$, $\frac{\partial \ell}{\partial w_{12}^2}$, $\frac{\partial \ell}{\partial w_{21}^2}$, $\frac{\partial \ell}{\partial w_{22}^2}$, $\frac{\partial \ell}{\partial w_{11}^3}$, $\frac{\partial \ell}{\partial w_{12}^3}$, $\frac{\partial \ell}{\partial b_1^2}$, $\frac{\partial \ell}{\partial b_2^2}$, $\frac{\partial \ell}{\partial b_1^3}$.

Would suffice to compute $\frac{\partial \ell}{\partial z_1^3}$, $\frac{\partial \ell}{\partial z_1^2}$, $\frac{\partial \ell}{\partial z_2^2}$.



Let us now turn to training an MLP using the data. Let us suppose that our dataset is $\mathcal{D} = \langle (\mathbf{x}_i, y_i) \rangle_{i=1}^N$. In the toy example, $\mathbf{x}_i \in \mathbb{R}^2$ and $y_i \in \{0, 1\}$, where 0 is red and 1 is blue, say. We will minimize an objective function of the form:

$$\mathcal{L}(\mathbf{W}^2, \mathbf{b}^2, \mathbf{W}^3, \mathbf{b}^3; \mathcal{D}) = \sum_{i=1}^{N} \ell(\mathbf{x}_i, y_i; \mathbf{W}^2, \mathbf{b}^2, \mathbf{W}^3, \mathbf{b}^3),$$

where $\ell(\mathbf{x}_i, y_i; \mathbf{W}^2, \mathbf{b}^2, \mathbf{W}^3, \mathbf{b}^3)$ denotes the loss on a single data point, as a result of the difference between the model prediction and the observed y_i. Let $\hat{y}_i$ denote the model prediction given the parameters and $\mathbf{x}_i$. In this case, $\hat{y}_i$ is the probability that the model believes the label should be 1. For classification problems, such as this, we can use the cross-entropy loss function,

$$\ell(\mathbf{x}_i, y_i; \mathbf{W}^2, \mathbf{b}^2, \mathbf{W}^3, \mathbf{b}^3) = -\left(y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)\right)$$

Compute gradients on a toy example

For classification problems, such as this, we can use the cross-entropy loss function,

$$\ell(\mathbf{x}_i, y_i; \mathbf{W}^2, \mathbf{b}^2, \mathbf{W}^3, \mathbf{b}^3) = -\left(y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)\right)$$

Let us compute the following:

1. $\frac{\partial \ell}{\partial a_1^3}$
2. $\frac{\partial a_1^3}{\partial z_1^3}$
3. $\frac{\partial z_1^3}{\partial \mathbf{a}^2}$
4. $\frac{\partial \mathbf{a}^2}{\partial \mathbf{z}^2}$

Compute gradients on a toy example
For classification problems, such as this, we can use the cross-entropy loss function,
$\ell(\mathbf{x}_i, y_i; \mathbf{W}^2, \mathbf{b}^2, \mathbf{W}^3, \mathbf{b}^3) = -\left(y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)\right)$
Let us compute the following:
1. $\frac{\partial \ell}{\partial a_1^3}$
2. $\frac{\partial a_1^3}{\partial z_1^3}$
3. $\frac{\partial z_1^3}{\partial \mathbf{a}^2}$
4. $\frac{\partial \mathbf{a}^2}{\partial \mathbf{z}^2}$

Let us denote by θ all the parameters in the model, $\mathbf{W}^2, \mathbf{b}^2, \mathbf{W}^3, \mathbf{b}^3$. Then we wish to find, $\frac{\partial \mathcal{L}}{\partial \theta}$. We can express this as,

$$\frac{\partial \mathcal{L}}{\partial \theta} = \sum_{i=1}^{N} \frac{\partial \ell(\mathbf{x}_i, y_i; \theta)}{\partial \theta}$$

Thus, in order to obtain the gradient of the objective function, $\mathcal{L}$, with respect to the parameters, we need to obtain the gradient of the loss function, ℓ, for a single datapoint. Note that other aspects of the model, such as the number of layers, the number of units in each layer and the activation functions are not considered to be parameters for the purposes of training. These could be treated as hyperparameters and model selection performed to select them. However, as neural networks take a rather long time to train, proper model selection is rather costly and some standard default parameters may be used.

Compute gradients on a toy example

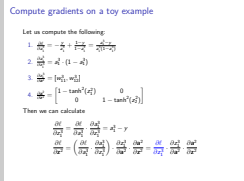
Let us compute the following:

1. $\frac{\partial \ell}{\partial a_1^3} = -\frac{y}{a_1^3} + \frac{1-y}{1-a_1^3} = \frac{a_1^3 - y}{a_1^3(1-a_1^3)}$
2. $\frac{\partial a_1^3}{\partial z_1^3} = a_1^3 \cdot (1 - a_1^3)$
3. $\frac{\partial z_1^3}{\partial \mathbf{a}^2} = [w_{11}^3, w_{12}^3]$
4. $\frac{\partial \mathbf{a}^2}{\partial \mathbf{z}^2} = \begin{bmatrix} 1 - \tanh^2(z_1^2) & 0 \\ 0 & 1 - \tanh^2(z_2^2) \end{bmatrix}$

Then we can calculate

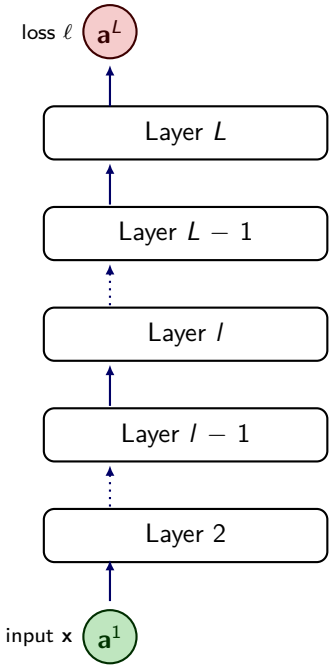
$$\frac{\partial \ell}{\partial z_1^3} = \frac{\partial \ell}{\partial a_1^3} \cdot \frac{\partial a_1^3}{\partial z_1^3} = a_1^3 - y$$

$$\frac{\partial \ell}{\partial \mathbf{z}^2} = \left(\frac{\partial \ell}{\partial a_1^3} \cdot \frac{\partial a_1^3}{\partial z_1^3} \right) \cdot \frac{\partial z_1^3}{\partial \mathbf{a}^2} \cdot \frac{\partial \mathbf{a}^2}{\partial \mathbf{z}^2} = \frac{\partial \ell}{\partial z_1^3} \cdot \frac{\partial z_1^3}{\partial \mathbf{a}^2} \cdot \frac{\partial \mathbf{a}^2}{\partial \mathbf{z}^2}$$



Although we could have computed the partial derivatives with respect to all the model parameters in an *ad hoc* fashion, let's do it using matrix operations and the following chain rule of multivariate calculus: Suppose $f : \mathbb{R}^n \to \mathbb{R}^k$, $g : \mathbb{R}^k \to \mathbb{R}^m$, let $h = g \circ f$, let $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{z} = f(\mathbf{x})$, then: $\frac{\partial h}{\partial \mathbf{x}} = \frac{\partial h}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial \mathbf{x}}$. Note that $\frac{\partial h}{\partial \mathbf{z}}$ is the $m \times k$ Jacobian matrix and $\frac{\partial \mathbf{z}}{\partial \mathbf{x}}$ is the $k \times n$ Jacobian matrix, the product of which gives the $m \times n$ Jacobian matrix $\frac{\partial h}{\partial \mathbf{x}}$.

This shows how we can compute the partial derivatives of the loss with respect to the parameters for a single datapoint. If using gradient descent, we need to do this for every datapoint in the training data and then add (or average) the derivatives, before performing a gradient step to update the model parameters. As is more common in the context of neural networks, we average the partial derivatives over a mini-batch rather than the entire training set.



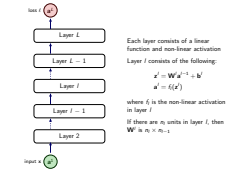
Each layer consists of a linear function and non-linear activation

Layer l consists of the following:

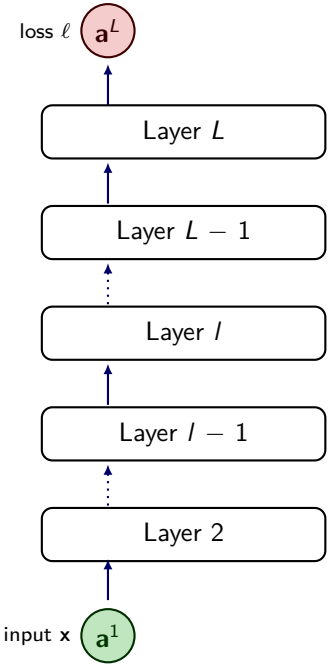
$$\mathbf{z}^l = \mathbf{W}^l \mathbf{a}^{l-1} + \mathbf{b}^l$$
$$\mathbf{a}^l = f_l(\mathbf{z}^l)$$

where f_l is the non-linear activation in layer l

If there are n_l units in layer l, then $\mathbf{W}^l$ is $n_l \times n_{l-1}$



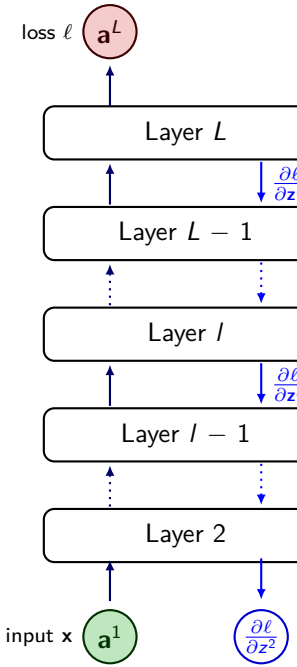
Let us consider a general neural network with L layers, the first being the inputs, $x_1, ..., x_D$ and the last layer being the output. Let $\mathbf{W}^i$, $\mathbf{b}^i$ for $i = 2, ..., L$ denote the weights between layers $i-1$ and i and the bias terms for the units in layer i respectively. Let $\mathbf{W}^{2:L}$ and $\mathbf{b}^{2:L}$ denote all the weights and biases in the neural network. Let us now look at a method to compute the gradient of the loss function $\ell(\mathbf{x}, y|\mathbf{W}^{2:L}, \mathbf{b}^{2:L})$ for a single training datapoint $(\mathbf{x}, y)$. Typically, we can think of y as representing a single value, a real number in case of regression, or a class label in the case of classification. However, sometimes for classification problems it may be more convenient to view y as a vector in C (number of classes) dimensions, where the class label is represented using a one-hot encoding. We will denote the neural network schematically as shown in the left Figure. For now, let us assume that all the layers are fully connected layers, i.e., every unit in layer $l-1$ has a connection to every unit layer l. Other architectures are possible and in fact widely used; once one has understood how to derive the forward and backward equations for models with fully connected layers, it is relatively straightforward to generalize to other architectures.



Forward Equations

1. $\mathbf{a}^1 = \mathbf{x}(\text{input})$
2. $\mathbf{z}^l = \mathbf{W}^l \mathbf{a}^{l-1} + \mathbf{b}^l$
3. $\mathbf{a}^l = f_l(\mathbf{z}^l)$
4. $\ell(\mathbf{a}^L, \mathbf{y})$

Let us suppose that we have some value for the parameters $\mathbf{W}^{2:L}, \mathbf{b}^{2:L}$ of the model. If we are just beginning the optimization, these will be initialized randomly. If we've already performed a few iterations of some iterative optimization algorithm, then the parameters will have been updated accordingly. The forward equations show how the predictions are made using the model, i.e., given the model parameters and some input $\mathbf{x}$, the output of the last layer $\mathbf{a}^L$ is used to make the prediction. For regression problems, $\mathbf{a}^L$ will typically be a single real number which is the predicted value. For classification problems, $\mathbf{a}^L$ will typically be a probability distribution over the C classes, with a^L_c representing the probability according to the model that the input $\mathbf{x}$ belongs to class c. (For the special case of binary classification, we'll typically assume that $\mathbf{a}^L$ is a single real number in $[0, 1]$ representing the probability that the input $\mathbf{x}$ has label 1 according to the model.)



Backward pass to compute derivatives

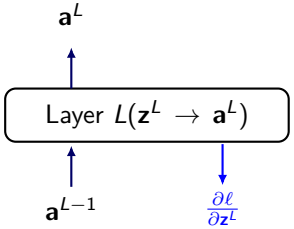
We'll use the convention that for a vector $\mathbf{z} \in \mathbb{R}^n$ and a function $f : \mathbb{R}^n \to \mathbb{R}$, where f takes values in $\mathbb{R}$, $\frac{\partial f}{\partial \mathbf{z}} \in \mathbb{R}^n$ is the *row vector* given by: $\frac{\partial f}{\partial \mathbf{z}} = \left[\frac{\partial f}{\partial z_1}, \frac{\partial f}{\partial z_2}, ..., \frac{\partial f}{\partial z_2}\right]$. If $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ is a function, with $(\mathbf{f}(\mathbf{z}))_i = f_i(\mathbf{z})$, then $\frac{\partial \mathbf{f}}{\partial \mathbf{z}}$ is the $m \times n$ Jacobian matrix, given by

$$\frac{\partial \mathbf{f}}{\partial \mathbf{z}} = \begin{bmatrix} \frac{\partial f_1}{\partial z_1} & \frac{\partial f_1}{\partial z_2} & \cdots & \frac{\partial f_1}{\partial z_n} \\ \frac{\partial f_2}{\partial z_1} & \frac{\partial f_2}{\partial z_2} & \cdots & \frac{\partial f_2}{\partial z_n} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial f_m}{\partial z_1} & \frac{\partial f_m}{\partial z_2} & \cdots & \frac{\partial f_m}{\partial z_n} \end{bmatrix}$$

Finally, if $f : \mathbb{R}^{n \times m} \to \mathbb{R}$, then for $\mathbf{W} \in \mathbb{R}^{n \times m}$, $\frac{\partial f}{\partial \mathbf{W}} \in \mathbb{R}^{n \times m}$ given by

$$\frac{\partial f}{\partial \mathbf{W}} = \begin{bmatrix} \frac{\partial f}{\partial W_{11}} & \frac{\partial f}{\partial W_{12}} & \cdots & \frac{\partial f}{\partial W_{1m}} \\ \frac{\partial f}{\partial W_{21}} & \frac{\partial f}{\partial W_{22}} & \cdots & \frac{\partial f}{\partial W_{2m}} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial f}{\partial W_{n1}} & \frac{\partial f}{\partial W_{n2}} & \cdots & \frac{\partial f}{\partial W_{nm}} \end{bmatrix}$$

Back propagation: output layer



$$\mathbf{z}^L = \mathbf{W}^L \mathbf{a}^{L-1} + \mathbf{b}^L$$

$$\mathbf{a}^L = f_L(\mathbf{z}^L)$$

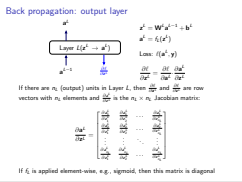
Loss: $\ell(\mathbf{a}^L, \mathbf{y})$

$$\frac{\partial \ell}{\partial \mathbf{z}^L} = \frac{\partial \ell}{\partial \mathbf{a}^L} \cdot \frac{\partial \mathbf{a}^L}{\partial \mathbf{z}^L}$$

If there are n_L (output) units in Layer L, then $\frac{\partial \ell}{\partial \mathbf{a}^L}$ and $\frac{\partial \ell}{\partial \mathbf{z}^L}$ are row vectors with n_L elements and $\frac{\partial \mathbf{a}^L}{\partial \mathbf{z}^L}$ is the $n_L \times n_L$ Jacobian matrix:

$$\frac{\partial \mathbf{a}^L}{\partial \mathbf{z}^L} = \begin{bmatrix} \frac{\partial a_1^L}{\partial z_1^L} & \frac{\partial a_1^L}{\partial z_2^L} & \cdots & \frac{\partial a_1^L}{\partial z_{n_L}^L} \\ \frac{\partial a_2^L}{\partial z_1^L} & \frac{\partial a_2^L}{\partial z_2^L} & \cdots & \frac{\partial a_2^L}{\partial z_{n_L}^L} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial a_{n_L}^L}{\partial z_1^L} & \frac{\partial a_{n_L}^L}{\partial z_2^L} & \cdots & \frac{\partial a_{n_L}^L}{\partial z_{n_L}^L} \end{bmatrix}$$

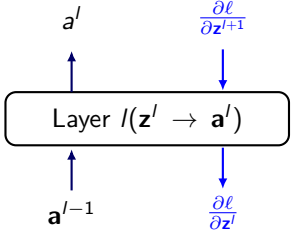
If f_L is applied element-wise, e.g., sigmoid, then this matrix is diagonal



When the model is being trained using some gradient based method, the parameters of the model are updated using the gradients of the loss function on the training data. For any individual datapoint $(\mathbf{x}, y)$, we'll denote the loss on that datapoint using $\ell(\mathbf{x}, y|\mathbf{W}^{2:L}, \mathbf{b}^{2:L})$ to emphasize that the loss under consideration is for the single datapoint and is affected by the model parameters. Typically, given the model parameters, $\mathbf{W}^{2:L}, \mathbf{b}^{2:L}$ producing a prediction $\mathbf{a}^L$, on some input $\mathbf{x}$ with target output y, the loss is a function of $\mathbf{a}^L$ and y. For example, when $\mathbf{a}^L \in \mathbb{R}$ and using the squared error as a loss function, $\ell(\mathbf{x}, y|\mathbf{W}^{2:L}, \mathbf{b}^{2:L}) = (\mathbf{a}^L - y)^2$. For classification problems with $y \in 1, ..., C$, the cross entropy loss function is given by:

$$\ell(\mathbf{x}, y|\mathbf{W}^{2:L}, \mathbf{b}^{2:L}) = -\sum_{c=1}^{C} \mathbb{1}(y = c) \log\left(a_c^L\right).$$

Back propagation: hidden layer



$$\mathbf{a}^l$$

the inputs into layer $l+1$

$$\mathbf{z}^{l+1} = \mathbf{W}^{l+1}\mathbf{a}^l + \mathbf{b}^{l+1}$$

$w_{j,k}^{l+1}$ weight on connection from k-th unit in layer l to j-th unit in layer $l+1$

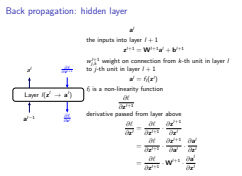
$$\mathbf{a}^l = f_l(\mathbf{z}^l)$$

f_l is a non-linearity function

$$\frac{\partial \ell}{\partial \mathbf{z}^{l+1}}$$

derivative passed from layer above

$$\begin{aligned}\frac{\partial \ell}{\partial \mathbf{z}^l} &= \frac{\partial \ell}{\partial \mathbf{z}^{l+1}} \cdot \frac{\partial \mathbf{z}^{l+1}}{\partial \mathbf{z}^l} \\ &= \frac{\partial \ell}{\partial \mathbf{z}^{l+1}} \cdot \frac{\partial \mathbf{z}^{l+1}}{\partial \mathbf{a}^l} \cdot \frac{\partial \mathbf{a}^l}{\partial \mathbf{z}^l} \\ &= \frac{\partial \ell}{\partial \mathbf{z}^{l+1}} \cdot \mathbf{W}^{l+1} \cdot \frac{\partial \mathbf{a}^l}{\partial \mathbf{z}^l}\end{aligned}$$



Given training data $\mathcal{D} = \langle(\mathbf{x}_i, y)\rangle_{i=1}^N$, the objective function wwe want to minimize as part of the training procedure is:

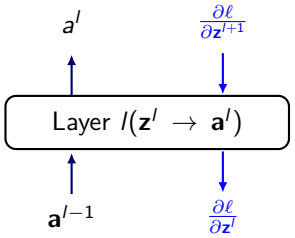
$$\mathcal{L}(\mathbf{W}^{2:L}, \mathbf{b}^{2:L}; \mathcal{D}) = \sum_{i=1}^{N} \ell(\mathbf{x}, y|\mathbf{W}^{2:L}, \mathbf{b}^{2:L})$$

For the optimization algorithm, we need to compute the gradient of $\mathcal{L}$ with respect to the parameters $\mathbf{W}^{2:L}, \mathbf{b}^{2:L}$,

$$\nabla_{\mathbf{W}^{2:L}, \mathbf{b}^{2:L}} \mathcal{L} = \sum_{i=1}^{N} \nabla_{\mathbf{W}^{2:L}, \mathbf{b}^{2:L}} \ell(\mathbf{x}, y|\mathbf{W}^{2:L}, \mathbf{b}^{2:L})$$

Thus the kep step in computing the gradient above is computing the gradient (partial derivatives) of $\ell(\mathbf{x}, y|\mathbf{W}^{2:L}, \mathbf{b}^{2:L})$ for a single datapoint; this is what the backpropagation algorithm does. If we use a mini-batch approach instead of taking the gradient over the entire dataset, we'll replace N by B, where B is the size of the mini-batch. It is important to shuffle the data before using batches and then cycle over the batches.

Gradients with respect to parameters



$$\mathbf{z}^l = \mathbf{W}^l \mathbf{a}^{l-1} + \mathbf{b}^l$$

$w^l_{j,k}$ weight on connection from k-th unit in layer $l-1$ to j-th unit in layer l

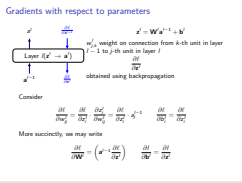
$\frac{\partial \ell}{\partial \mathbf{z}^l}$ obtained using backpropagation

Consider

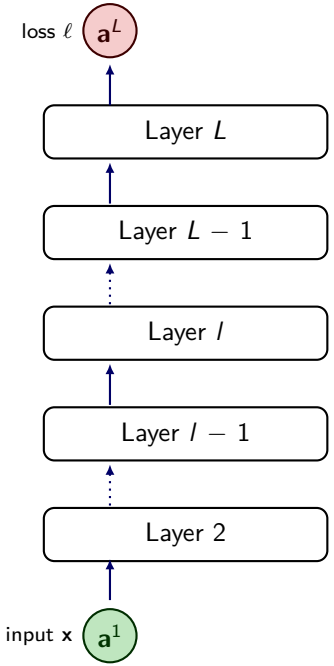
$$\frac{\partial \ell}{\partial w^l_{ij}} = \frac{\partial \ell}{\partial z^l_i} \cdot \frac{\partial z^l_i}{\partial w^l_{ij}} = \frac{\partial \ell}{\partial z^l_i} \cdot a^{l-1}_j \qquad \frac{\partial \ell}{\partial b^l_i} = \frac{\partial \ell}{\partial z^l_i}$$

More succinctly, we may write

$$\frac{\partial \ell}{\partial \mathbf{W}^l} = \left(\mathbf{a}^{l-1} \frac{\partial \ell}{\partial \mathbf{z}^l}\right) \qquad \frac{\partial l}{\partial \mathbf{b}^l} = \frac{\partial l}{\partial \mathbf{z}^l}$$

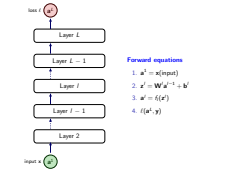


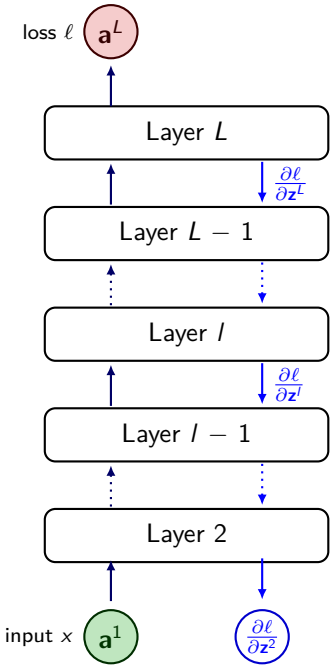
In order to compute the gradient $\nabla_{\mathbf{W}^{2:L},\mathbf{b}^{2:L}} \ell(\mathbf{x}, y | \mathbf{W}^{2:L}, \mathbf{b}^{2:L})$, we need to compute the partial derivatives $\frac{\partial \ell}{\partial \mathbf{W}^i}$ and $\frac{\partial \ell}{\partial i}$ for $i = 2, ..., L$ (in the lecture the loss function refers to a single datapoint $(\mathbf{x}, y)$ and the model parameters are $\mathbf{W}^{2:L}$, $\mathbf{b}^{2:L}$). In an intermediate step, we'll compute the partial derivatives $\frac{\partial \ell}{\partial \mathbf{z}^l}$.



Forward equations

1. $\mathbf{a}^1 = \mathbf{x}(\text{input})$
2. $\mathbf{z}^l = \mathbf{W}^l \mathbf{a}^{l-1} + \mathbf{b}^l$
3. $\mathbf{a}^l = f_l(\mathbf{z}^l)$
4. $\ell(\mathbf{a}^L, \mathbf{y})$



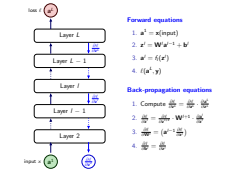


Forward equations

1. $\mathbf{a}^1 = \mathbf{x}(\text{input})$
2. $\mathbf{z}^l = \mathbf{W}^l \mathbf{a}^{l-1} + \mathbf{b}^l$
3. $\mathbf{a}^l = f_l(\mathbf{z}^l)$
4. $\ell(\mathbf{a}^L, \mathbf{y})$

Back-propagation equations

1. Compute $\frac{\partial \ell}{\partial \mathbf{z}^L} = \frac{\partial \ell}{\partial \mathbf{a}^L} \cdot \frac{\partial \mathbf{a}^L}{\partial \mathbf{z}^L}$
2. $\frac{\partial \ell}{\partial \mathbf{z}^l} = \frac{\partial \ell}{\partial \mathbf{z}^{l+1}} \cdot \mathbf{W}^{l+1} \cdot \frac{\partial \mathbf{a}^l}{\partial \mathbf{z}^l}$
3. $\frac{\partial \ell}{\partial \mathbf{W}^l} = \left(\mathbf{a}^{l-1} \frac{\partial \ell}{\partial \mathbf{z}^l}\right)$
4. $\frac{\partial l}{\partial \mathbf{b}^l} = \frac{\partial l}{\partial \mathbf{z}^l}$



The backpropagation algorithm, implemented by using the forward equations and the backward equations, is nothing but an application of the chain rule of multivariate calculus. If we represent the loss as a function of the weight parameters, there are many paths going from a weight W_{ij}^l to the loss function, through the various layers. The chain rule requires us to multiply all the partial derivatives across each of these paths and sum them up. The backpropagation algorithm is a way of managing these computations efficiently, by storing repeatedly used partial derivatives, i.e., the terms $\frac{\partial \ell}{\partial \mathbf{z}^l}$. Thus, we reduce the running time of the naive algorithm to compute the gradient at the cost of some extra space.

Computational questions

What is the running time to compute the gradient for a single data point?

- As many matrix multiplications as there are fully connected layers
- Performed twice during forward and backward pass

What is the space requirement?

- Need to store vectors $\mathbf{a}^l$, $\mathbf{z}^l$, and $\frac{\partial \ell}{\partial \mathbf{z}^l}$ for each layer

Can we process multiple examples together?

- Yes, if we minibatch, we perform tensor operations
- Make sure that all parameters fit in GPU memory

Computational questions
What is the running time to compute the gradient for a single data point?
- As many matrix multiplications as there are fully connected layers
- Performed twice during forward and backward pass
What is the space requirement?
- Need to store vectors $\mathbf{a}^l$, $\mathbf{z}^l$, and $\frac{\partial \ell}{\partial \mathbf{z}^l}$ for each layer
Can we process multiple examples together?
- Yes, if we minibatch, we perform tensor operations
- Make sure that all parameters fit in GPU memory

The dominant term in the running time of the backpropagation algorithm is the matrix multiplications that are required in the backward and forward equations. In addition to storing all the model parameters, $\mathbf{W}^{2:L}$, $\mathbf{b}^{2:L}$, we also need to store the pre-activations $\mathbf{z}^l$ and activations $\mathbf{a}^l$ at each layer.
When actually implementing the algorithm, rather than performing the gradient computation for the loss of a single datapoint, you would do it for a batch of examples together. The matrix operations in the above equations then need to be replaced by suitable tensor operations, where the extra dimension represents the different datapoints. While there is a saving to be obtained by processing multiple datapoints together, the batch size should be chosen to be suitably small so that all the computations can still be performed on the memory available on the GPU; otherwise the advantage of processing multiple examples simultaneously will be lost.

Outline

Neural Networks

Training Deep Neural Networks

Convolution Neural Networks

Recurrent Neural Networks

Transformers

Outline

Neural Networks

Training Deep Neural Networks

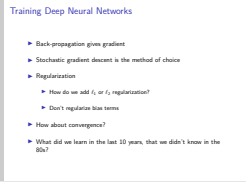
Convolution Neural Networks

Recurrent Neural Networks

Transformers

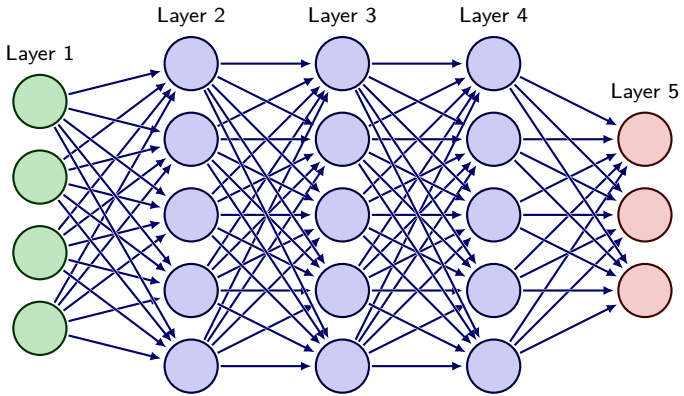
Training Deep Neural Networks

- Back-propagation gives gradient
- Stochastic gradient descent is the method of choice
- Regularization
  - How do we add ℓ_1 or ℓ_2 regularization?
  - Don't regularize bias terms
- How about convergence?
- What did we learn in the last 10 years, that we didn't know in the 80s?



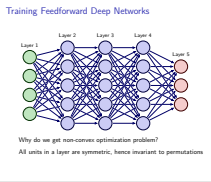
The backpropagation algorithm gives the gradient of the loss function with respect to a single datapoint. The gradient of the objective function can be computed by summing this over the entire dataset (or a minibatch). As with any other machine learning method, neural networks trained using backpropagation may be prone to overfitting. This is especially the case with very large neural networks, in which the number of parameters is typically much greater than the number of training examples available. Methods such as ℓ_2 and ℓ_1 regularisation which we used in the context of linear regression, logistic regression, etc. can also be used when training neural networks. In addition, there are a few other aspects of the training that are more specific to neural networks. Many of these ideas are to be viewed as "known hacks", or current best practice, rather than exact science.

Training Feedforward Deep Networks



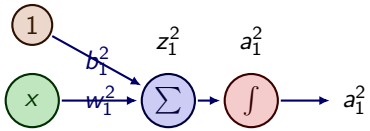
Why do we get non-convex optimization problem?

All units in a layer are symmetric, hence invariant to permutations



Training neural networks is an active area or research and these ideas may evolve over time. The main difficulty is that the objective function being minimised when training neural networks is not a convex function of the parameters, and hence the training procedure may at times feel more art than science. Before we discuss some methods to improve the training of neural networks, let us understand some of the problems that arise when attempting to train neural networks.

A toy example to illustrate saturation



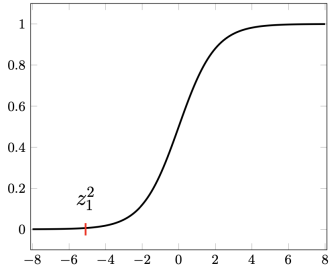
Target is $y = \frac{1-x}{2}$ and $x \in \{1, -1\}$

Squared Loss Function

$$\ell(a_1^2, y) = (a_1^2 - y)^2$$

$$\frac{\partial \ell}{\partial z_1^2} = 2(a_1^2 - y) \cdot \frac{\partial a_1^2}{\partial z_1^2} = 2(a_1^2 - y)\sigma'(z_1^2)$$

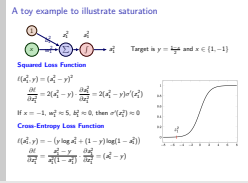
If $x = -1$, $w_1^2 \approx 5$, $b_1^2 \approx 0$, then $\sigma'(z_1^2) \approx 0$



Cross-Entropy Loss Function

$$\ell(a_1^2, y) = -\left(y \log a_1^2 + (1-y)\log(1-a_1^2)\right)$$

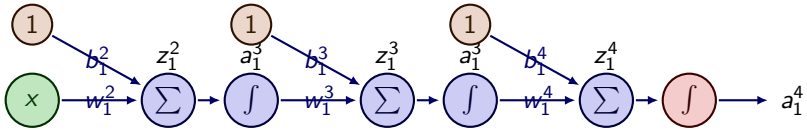
$$\frac{\partial \ell}{\partial z_1^2} = \frac{a_1^2 - y}{a_1^2(1-a_1^2)} \cdot \frac{\partial a_1^2}{\partial z_1^2} = (a_1^2 - y)$$



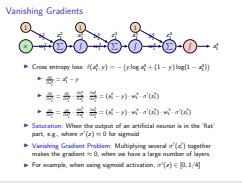
Suppose we wish to train a neural network with a single unit and the sigmoid activation function. The only parameters in the model are w_{11}^2 and b_1^2. Suppose we initialize with $w_{11}^2 = -5$ and $b_1^2 = 0$, and use the squared loss function. For $x = -1$, (the target is $y = 1$), although the loss $(a_1^2 - y)^2$ is very large, the derivative $\frac{\partial \ell}{\partial z_1^2} \approx 0$ as $\sigma'(z_1^2) \approx 0$. This happens because the sigmoid function is very flat at $z_1^2 = -5$ even though the prediction is off by a lot. The pre-activations being in the range where the activation function is very flat is referred to as *saturation*. When the neural networks are saturated, gradient steps may not make much progress (as the gradient is very small), even though the loss is large.

In this case, we can get around this problem by using the cross entropy loss function instead. For cross entropy loss, it is easy to show that when $|a_1^2 - y|$ is large, the magnitude of the gradient will be large as well.

Vanishing Gradients



- Cross entropy loss: $\ell(a_1^4, y) = -\left(y \log a_1^4 + (1-y)\log(1-a_1^4)\right)$
  - $\frac{\partial \ell}{\partial z_1^4} = a_1^4 - y$
  - $\frac{\partial \ell}{\partial z_1^3} = \frac{\partial \ell}{\partial z_1^4} \cdot \frac{\partial z_1^4}{\partial a_1^3} \cdot \frac{\partial a_1^4}{\partial z_1^3} = (a_1^4 - y) \cdot w_1^4 \cdot \sigma'(z_1^3)$
  - $\frac{\partial \ell}{\partial z_1^2} = \frac{\partial \ell}{\partial z_1^3} \cdot \frac{\partial z_1^3}{\partial a_1^2} \cdot \frac{\partial a_1^2}{\partial z_1^2} = (a_1^4 - y) \cdot w_1^4 \cdot \sigma'(z_1^3) \cdot w_1^3 \cdot \sigma'(z_1^2)$
- Saturation: When the output of an artificial neuron is in the 'flat' part, e.g., where $\sigma'(z) \approx 0$ for sigmoid
- Vanishing Gradient Problem: Multiplying several $\sigma'(z_i^l)$ together makes the gradient ≈ 0, when we have a large number of layers
- For example, when using sigmoid activation, $\sigma'(z) \in [0, 1/4]$



Let us consider the same problem but build a network with three hidden layers. The main purpose of these examples is to demonstrate the difficulties arising in training neural networks. (You would never use these kinds of networks for such simple problems.) Let us look at the derivative. We see that the derivative is a product of terms containing $\sigma'(z_1^3)$ and $\sigma'(z_1^2)$. Note that $\sigma'(t) = \sigma(t)\,(1-\sigma(t)) \leq \frac{1}{4}$ (For two numbers, $a, b \in [0,1]$ such that $a+b=1$, the maximum possible value for ab is $1/4$). The product of several numbers, each less than $1/4$, approaches 0 rather quickly. Thus, the derivative may *vanish*, this is referred to as the *vanishing gradient problem*.

It may be that the product of the weights w_1^4 and w_1^3 counteracts this effect, but unless these products exactly cancel each other, we may get a gradient that either *vanishes* or *explodes*. These problems are referred to as the exploding and vanishing gradient problems respectively.

Avoiding Saturation

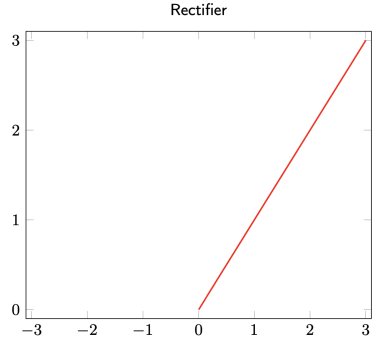
Use *rectified linear units*

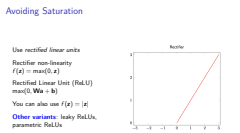
Rectifier non-linearity
$f(\mathbf{z}) = \max(0, \mathbf{z})$

Rectified Linear Unit (ReLU)
$\max(0, \mathbf{Wa} + \mathbf{b})$

You can also use $f(\mathbf{z}) = |\mathbf{z}|$

Other variants: leaky ReLUs, parametric ReLUs





The problem of saturation is inherent to many of the activations functions used in neural networks, such as sigmoid, tanh, etc. One activation function that has been widely used in the last few years, is the rectifier, $f(z) = \max(0, z)$. The corresponding unit is called a rectified linear unit, or ReLU for short. The rectifier activation function has the advantage that it only saturates on one side. Unless all the datapoints result in a negative preactivation, at least for some datapoints, the rectifier will always have derivative 1, and so to some extent the saturation, and hence vanishing gradient problem can be avoided.

Initializing Weights and Biases

Initializing is important when minimizing non-convex functions.
We may get very different results depending on where we start the optimization.
Suppose we were using a *sigmoid unit*, how would you initialize the weights?

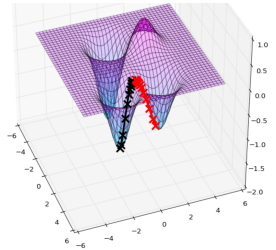
- Suppose $z = \sum_{i=1}^{D} w_i a_i$
- E.g., choose $w_i \in \left[-\frac{1}{\sqrt{D}}, \frac{1}{\sqrt{D}}\right]$ at random

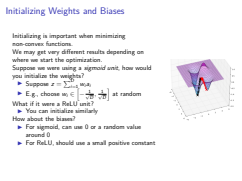
What if it were a ReLU unit?

- You can initialize similarly

How about the biases?

- For sigmoid, can use 0 or a random value around 0
- For ReLU, should use a small positive constant





Initialisation is also important to make sure that the network is not in a saturated state at the beginning of optimisation. Suppose $w_1, ..., w_D$ are the weights going into a sigmoid unit, then it is usually a good idea to set the initial values drawn randomly from $\mathcal{N}(0, \sigma^2), \sigma^2 = 1/D$, assuming that the inputs x_i, themselves satisfy, $E[x_2^i] \approx 1$. For *sigmoid* units the bias term can be set to 0, or some small random number (positive for negative). For rectified linear units, it's a good idea to set the bias to be a small positive number, so that most units are not saturated to begin with. The weights for the rectified linear unit can be set as in the case of *sigmoid* units.

It is important to stress that the randomness in the initializations plays an important role in symmetry breaking. All the units in a given layer typically are symmetric to begin with. If all the weights are initialized identically, then there will be nothing to differentiate them during training. This is very bad because all the units are computing exactly the same thing and it is unlikely that such a neural network would have good performance.

Avoiding Overfitting

Deep Neural Networks have a lot of parameters

- Fully connected layers with $n_1, n_2, ..., n_L$ units have at least $n_1 n_2 + n_2 n_3 + ... + n_{L-1} n_L$ parameters
- A typical training of an MLP for digit recognition with 2 million parameters and only 60,000 training images
- For image detection, one of the most famous models, the neural net used by Krizhevsky, Sutskever, Hinton (2012) has 60 million parameters and 1.2 million training images
- How do we prevent deep neural networks from overfitting?

Avoiding Overfitting
Deep Neural Networks have a lot of parameters
- Fully connected layers with $n_1, n_2, ..., n_L$ units have at least $n_1 n_2 + n_2 n_3 + ... + n_{L-1} n_L$ parameters
- A typical training of an MLP for digit recognition with 2 million parameters and only 60,000 training images
- For image detection, one of the most famous models, the neural net used by Krizhevsky, Sutskever, Hinton (2012) has 60 million parameters and 1.2 million training images
- How do we prevent deep neural networks from overfitting?

The number of parameters in neural networks used in practice can be pretty large. It is not a surprise then that these networks overfit unless specific techniques are used to prevent it. Classical methods of regularisation such as an ℓ_1 or ℓ_2 penalty can be used. There are a few other approaches specific to neural networks that we'll discuss here.

Early Stopping

Maintain validation set and stop training when error on validation set stops decreasing

What are the computational costs?

- Need to compute validation error
- Can do this every few iterations to reduce overhead

What are the advantages?

- If validation error flattens or starts increasing, can stop optimization
- Prevents overfitting

See paper by Hardt, Recht and Singer (2015)

Early Stopping
Maintain validation set and stop training when error on validation set stops decreasing
What are the computational costs?
Need to compute validation error
Can do this every few iterations to reduce overhead
What are the advantages?
If validation error flattens or starts increasing, can stop optimization
Prevents overfitting
See paper by Hardt, Recht and Singer (2015)

Almost always, neural networks are trained using iterative optimisation methods. One way to reduce overfitting is to run the optimisation for a relatively small number of steps. This is referred to as *early stopping*. A principled way to decide when to stop is to keep aside a validation set and measure the performance of the classifier after each gradient step on it. Of course, the optimization algorithm should only use the training set, not the validation set. Once the performance on the validation set starts plateauing, the optimization procedure can be stopped. One thing to bear in mind is that because this is a non-convex optimization problem, plateauing does not necessarily mean being close to optimality. This may be especially the case for really large networks and if the landscape of the objective function itself has plateaus. This may simply mean that we are stuck at a "local minimum", rather than indicating overfitting. Changing the learning rate in such cases can be used as a way to see if the training procedure can escape the local minimum.

Add Data: Modified Data

Typically, getting additional data is either impossible or expensive

Fake the data!

Images can be translated slightly, rotated slightly, change of brightness, etc

Google Offline Translate trained on entirely fake data!

[1] Google Research Blog: https://blog.research.google/2015/07/how-google-translate-squeezes-deep.html

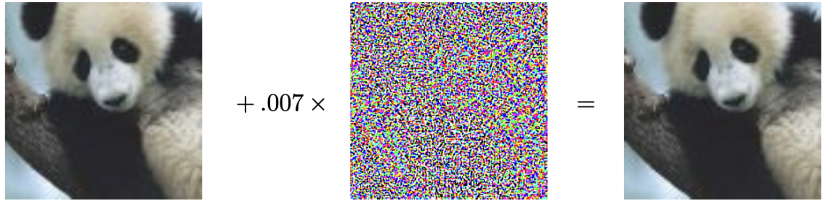
Add Data: Modified Data
Typically, getting additional data is either impossible or expensive
Fake the data!
Images can be translated slightly, rotated slightly, change of brightness, etc
Google Offline Translate trained on entirely fake data!

The most obvious method to reduce overfitting is to increase the quantity of training data. Of course, this is easier said than done. Obtaining additional data may be at best expensive and at worst impossible. In some domains, tricks can be used that allow us to add 'fake' data based on the existing training data. For example, when using neural networks for object detection, minor rotations, translations, etc. do not affect whether or not an image contains a coffee cup, or a dog, or any other thing. Thus, we may get more data by merely modifying existing training data. An extreme example of this is a Google translation app that is trained using entirely fake data; the only requirement in this case was to recognise letters of the alphabet and they started by generating such images and modifying them through simple transformations. This simple approach of augmenting data does not work when we have no obvious elementary transformations that we can apply to data and be sure that the targets remain unchanged.

Add Data: Adversarial Training

Take trained (or partially trained model)

Create examples by modifications "imperceptible to the human eye", but where the model fails

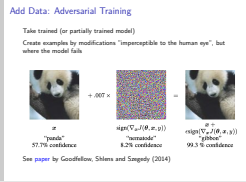


$\boldsymbol{x}$
"panda"
57.7% confidence

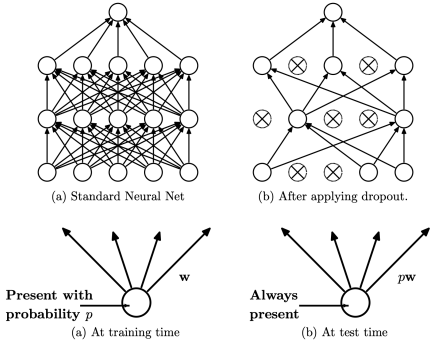
$\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
"nematode"
8.2% confidence

$\boldsymbol{x} + \epsilon \text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
"gibbon"
99.3 % confidence

See paper by Goodfellow, Shlens and Szegedy (2014)



Dropout



(a) Standard Neural Net (b) After applying dropout.

(a) At training time (b) At test time

- For input $\mathbf{x}$ drop each hidden unit with probability $1/2$ independently
- Every input will have a potentially different mask
- Potentially exponentially different models, but have "same weights"
- After training whole network is used by having all the weights

See paper by Srivastava, Hinton, Krizhevsky (2014)

Dropout
- For input x drop each hidden unit with probability 1/2 independently
- Every input will have a potentially different mask
- Potentially exponentially different models, but have "same weights"
- After training whole network is used by having all the weights

See paper by Srivastava, Hinton, Krizhevsky (2014)

The basic idea of Dropout is the following. During each training step (i.e., gradient update step), a fraction (typically half) of the units in specified hidden layers are "dropped". Thus, only the weights and biases related to units that are not dropped are udpated in this gradient step. However, the choice of units to be dropped is random and different at each gradient update step. The intuition behind this is that it prevents co-adaptation among different neurons. At test time, the entire network is used. For this reason, all the weights need to be scaled appropriately before using the model (the weights need to be halved, if the dropout rate was $1/2$).

The dropout approach can be viewed as a form of model averaging. In this sense, it is connected to an old idea from statistics, called bootstrap aggregation, or bagging.

Other Ideas to Reduce Overfitting

- Hard constraints on weights
- Gradient Clipping
- Inject noise into the system
- Unsupervised Pre-training
- Enforce sparsity in the neural network

Other Ideas to Reduce Overfitting
- Hard constraints on weights
- Gradient Clipping
- Inject noise into the system
- Unsupervised Pre-training
- Enforce sparsity in the neural network

Outline

Neural Networks

Training Deep Neural Networks

Convolution Neural Networks

Recurrent Neural Networks

Transformers

Outline

Neural Networks

Training Deep Neural Networks

Convolution Neural Networks

Recurrent Neural Networks

Transformers

Convolutional Neural Networks (convnets)

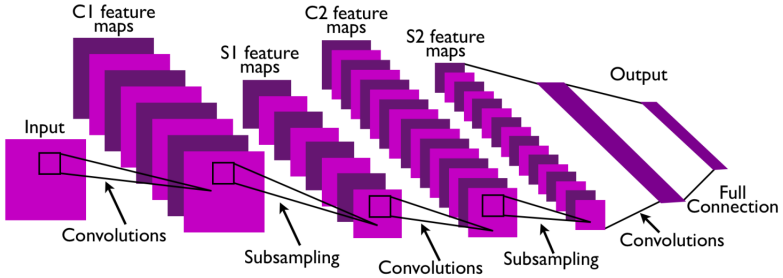
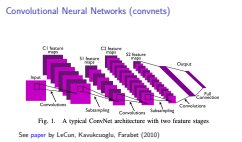


Fig. 1. A typical ConvNet architecture with two feature stages

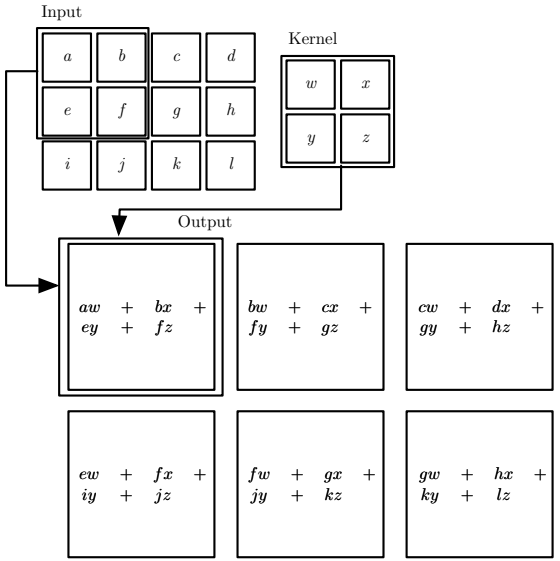
See paper by LeCun, Kavukcuoglu, Farabet (2010)

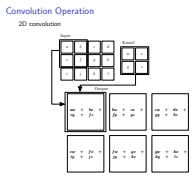


Convolutional neural networks, *convnets* for short, are a class of neural networks that have enjoyed great practical success in recent years. Convolutional neural networks can be viewed as a way of reducing overfitting by introducing weight-tying that exploits the geometry. The first and most successful application of convolutional neural networks was in the context of image data, however, of late they have been employed successfully on audio, video and even text data.

Convolution Operation

2D convolution





Convolution leverages several important ideas that can help improve a machine learning system, e.g., *sparse interactions*, *parameter sharing*.

Convolutional networks typically have sparse interactions, which is accomplished by making the kernel smaller than the input. For example, when processing an image, the input image might have thousands or millions of pixels, but we can detect small, meaningful features such as edges with kernels that occupy only tens or hundreds of pixels.

Parameter sharing refers to using the same parameter for more than one function in a model. Ina convolutional neural net, each member of the kernel is used at every position of the input (except perhaps some of the boundary pixels, depending on the design decisions regarding the boundary). The parameter sharing used by the convolution operation means that rather than learning a separate set of parameters for every location, we learn only one set.

Convolution

In general, a convolution filter f is a tensor of dimension $W_f \times H_f \times F_l$, where F_l is the number of channels in the previous layer

Strides in x and y directions dictate which convolutions are computed to obtain the next layer

Zero-padding can be used if required to adjust layer sizes and boundaries

Typically, a convolution layer will have a large number of filters, the number of channels in the next layer will be the same as the number of filters used

Convolution

In general, a convolution filter f is a tensor of dimension $W_f \times H_f \times F_l$, where F_l is the number of channels in the previous layer

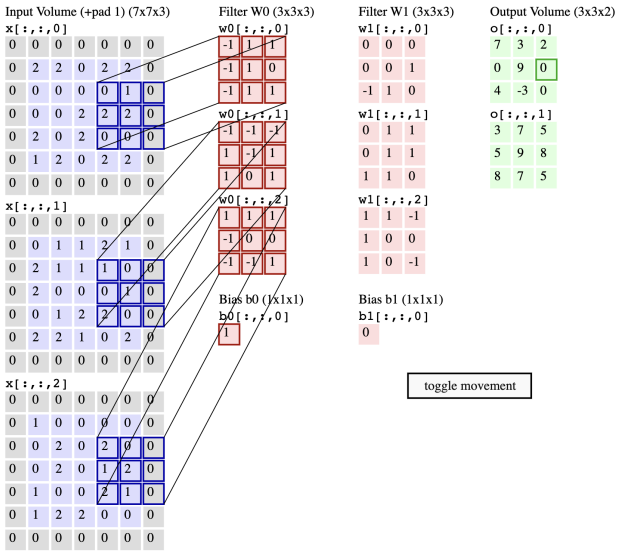
Strides in x and y directions dictate which convolutions are computed to obtain the next layer

Zero-padding can be used if required to adjust layer sizes and boundaries

Typically, a convolution layer will have a large number of filters, the number of channels in the next layer will be the same as the number of filters used

Image Convolution

Two filters, zero padding, stride one[2]



[2] https://cs231n.github.io/convolutional-networks/

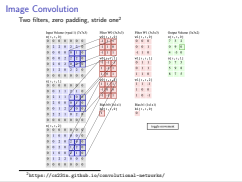
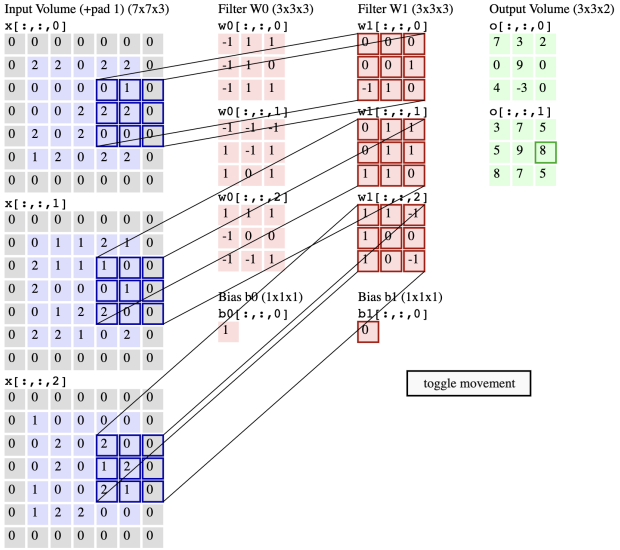
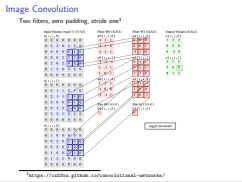


Image Convolution

Two filters, zero padding, stride one[3]

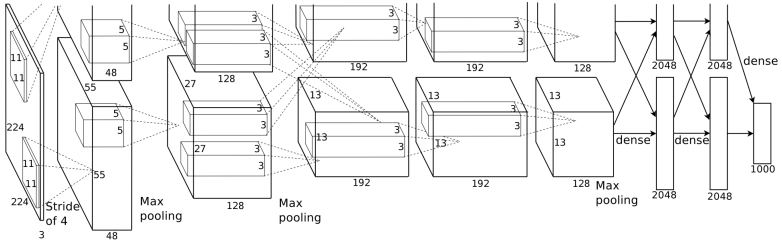


[3] https://cs231n.github.io/convolutional-networks/

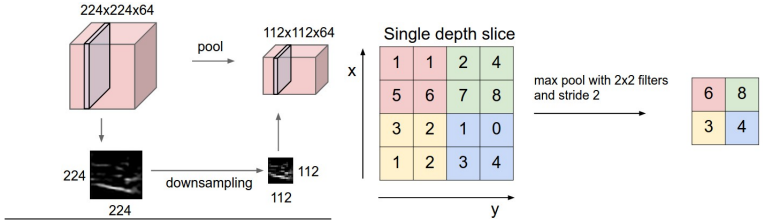


Deep Convnets

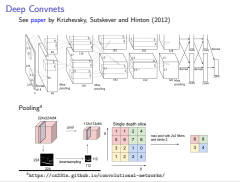
See paper by Krizhevsky, Sutskever and Hinton (2012)



Pooling[4]



[4] https://cs231n.github.io/convolutional-networks/



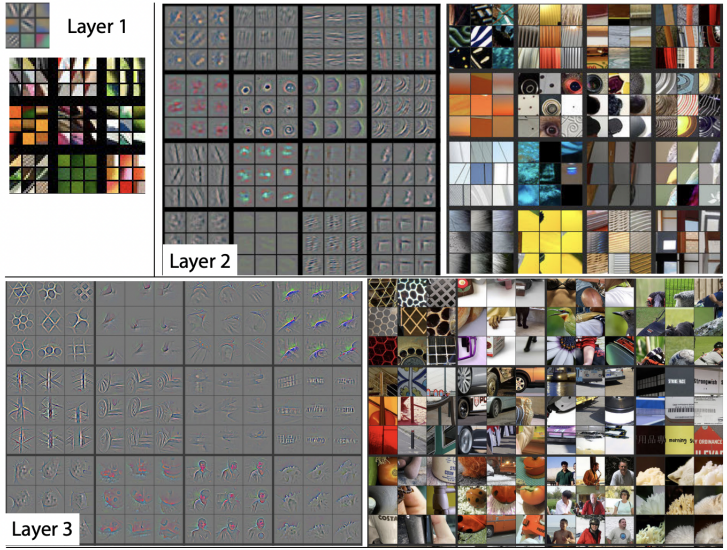
Convolutional neural networks frequently make use of a pooling operation after convolution layers. In all cases, pooling helps to make the representation approximately invariant to small translations of the input. Invariance to translation means that if we translate the input by a small amount, the values of most of the pooled outputs do not change.

A pooling function replaces the output of the net at a certain location with a summary statistic of the nearby outputs. For example, the max pooling operation reports the maximum output within a rectangular neighborhood. Mathematically, we can express the forward and backward equations for pooling operations as follows

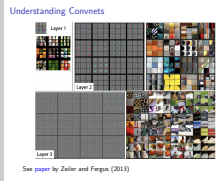
$$s^{l+1}_{i',j'} = \max_{i,j \in \Omega(i',j')} a^l_{i,j} \qquad \frac{\partial s^{l+1}_{i',j'}}{\partial a^l_{i,j}} = \mathbb{1}\left((i,j) = \arg\max_{i'',j'' \in \Omega(i',j')} a^l_{i'',j''}\right)$$

These local equations can be easily integrated into the general forward an backward equations for the whole network.

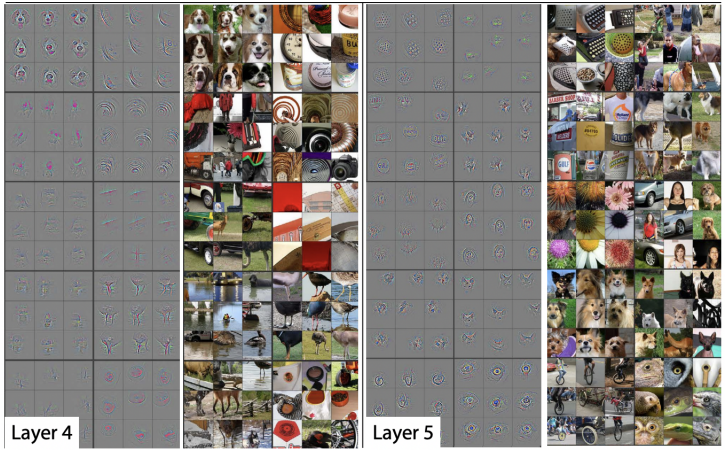
Understanding Convnets



See paper by Zeiler and Fergus (2013)



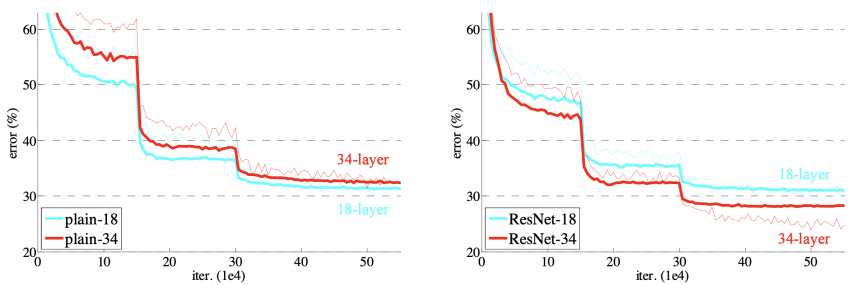
Understanding Convnets



See paper by Zeiler and Fergus (2013)

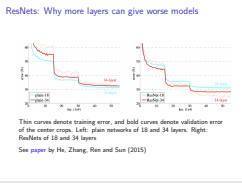
Understanding Convnets
See paper by Zeiler and Fergus (2013)

ResNets: Why more layers can give worse models



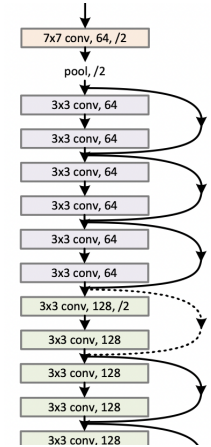
Thin curves denote training error, and bold curves denote validation error of the center crops. Left: plain networks of 18 and 34 layers. Right: ResNets of 18 and 34 layers

See paper by He, Zhang, Ren and Sun (2015)

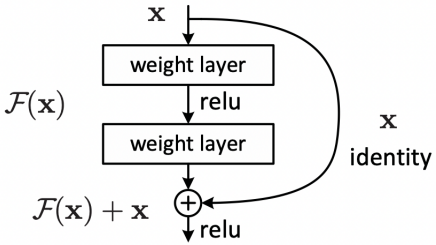


ResNets

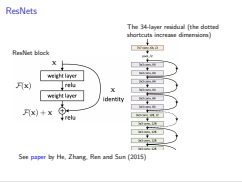
The 34-layer residual (the dotted shortcuts increase dimensions)



ResNet block



See paper by He, Zhang, Ren and Sun (2015)



Outline

Neural Networks

Training Deep Neural Networks

Convolution Neural Networks

Recurrent Neural Networks

Transformers

Outline
Neural Networks
Training Deep Neural Networks
Convolution Neural Networks
Recurrent Neural Networks
Transformers

Language Models

A language model assigns a probability to a sequence of words, such that $\sum_{w \in \Sigma^*} p(w) = 1$

Given the observed training text, how probable is this new utterance?

Thus we can compare different orderings of words (e.g. Translation):

$$p(\text{he likes apples}) > p(\text{apples likes he})$$

or choice of words (e.g., Speech Recognition)

$$p(\text{he likes apples}) > p(\text{he licks apples})$$

Most language models employ the chain rule to decompose the joint probability into a sequence of conditional probabilities:

$$p(w_1, w_2, w_3, ..., w_N) = p(w_1)p(w_2|w_1)p(w_3|w_1, w_2) \cdots \cdots p(w_N|w_1, w_2, ..., w_{N-1})$$

Note that this decomposition is exact and allows us to model complex joint distributions by learning conditional distributions over the next word (w_n) given the history of words observed $(w_1, w_2, ..., w_{n-1})$

Language Models
A language model assigns a probability to a sequence of words, such that $\sum_{w \in \Sigma^*} p(w) = 1$
Given the observed training text, how probable is this new utterance?
Thus we can compare different orderings of words (e.g. Translation):
$p(\text{he likes apples}) > p(\text{apples likes he})$
or choice of words (e.g., Speech Recognition)
$p(\text{he likes apples}) > p(\text{he licks apples})$
Most language models employ the chain rule to decompose the joint probability into a sequence of conditional probabilities
$p(w_1, w_2, w_3, ..., w_N) = p(w_1)p(w_2|w_1)p(w_3|w_1, w_2) \cdots \cdots p(w_N|w_1, w_2, ..., w_{N-1})$
Note that this decomposition is exact and allows us to model complex joint distributions by learning conditional distributions over the next word (w_n) given the history of words observed $(w_1, w_2, ..., w_{n-1})$

Language Models

The simple objective of modelling the next word given the observed history contains much of the complexity of natural language understanding.

Consider predicting the extension of the utterance:

$$p(\cdot|\text{There she built a })$$

With more context we are able to use our knowledge of both language and the world to heavily constrain the distribution over the next word:

$$p(\cdot|\text{Alice went to the beach. There she built a })$$

There is evidence that human language acquisition partly relies on future prediction

Language modelling is a time series prediction problem in which we must be careful to train on the past and test on the future

With Recurrent Neural Networks we compress the entire history in a fixed length vector, enabling long range correlations to be captured.

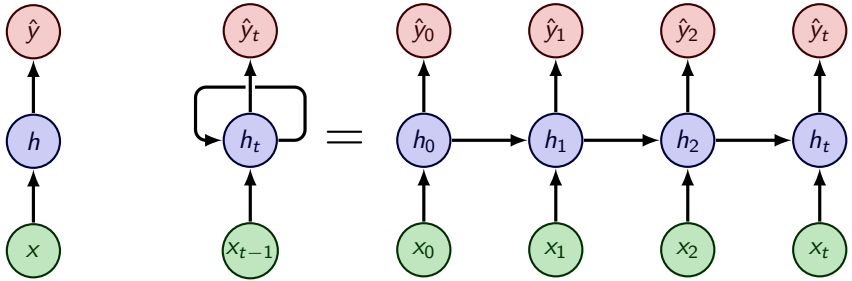
Language Models
The simple objective of modelling the next word given the observed history contains much of the complexity of natural language understanding.
Consider predicting the extension of the utterance:
$p(\cdot|\text{There she built a })$
With more context we are able to use our knowledge of both language and the world to heavily constrain the distribution over the next word:
$p(\cdot|\text{Alice went to the beach. There she built a })$
There is evidence that human language acquisition partly relies on future prediction
Language modelling is a time series prediction problem in which we must be careful to train on the past and test on the future
With Recurrent Neural Networks we compress the entire history in a fixed length vector, enabling long range correlations to be captured.

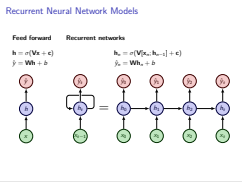
Recurrent Neural Network Models

Feed forward **Recurrent networks**

$\mathbf{h} = \sigma(\mathbf{V}\mathbf{x} + \mathbf{c})$ $\mathbf{h}_n = \sigma(\mathbf{V}[\mathbf{x}_n; \mathbf{h}_{n-1}] + \mathbf{c})$

$\hat{y} = \mathbf{W}\mathbf{h} + b$ $\hat{y}_n = \mathbf{W}\mathbf{h}_n + b$



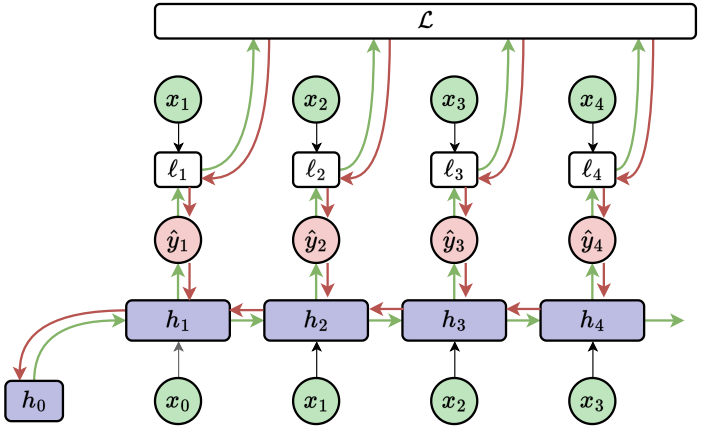


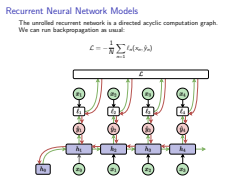
We can conceptually think of a recurrent network as a feed forward network where the hidden layer is a function of both the input $\mathbf{x}$ and the previous hidden layer $\mathbf{h}$. When we apply an RNN to a sequence we can view it as one very deep feed forward network. with output layers attached to every hidden layer. In this form an RNN is referred to as *unrolled*. The unrolled recurrent network is a directed acyclic computation graph. In principle we can run backpropagation as usual. Ithough this may not always be possible computationally. The application of backprogration to unrolled recurrent networks is called the Backpropagation Through Time (BPTT) algorithm.

Recurrent Neural Network Models

The unrolled recurrent network is a directed acyclic computation graph.
We can run backpropagation as usual:

$$\mathcal{L} = -\frac{1}{N}\sum_{n=1} \ell_n(x_n, \hat{y}_n)$$

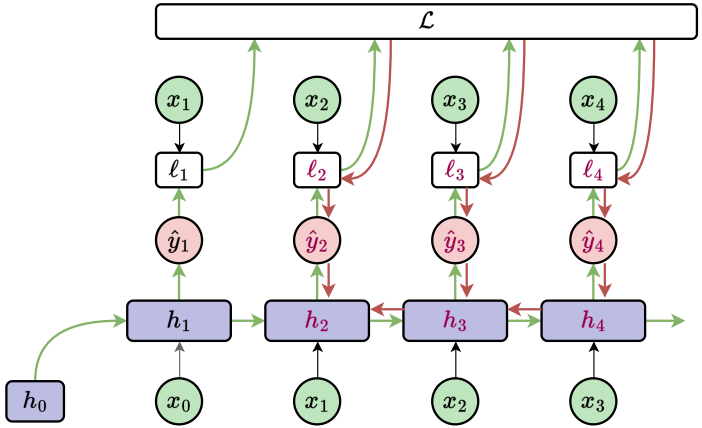


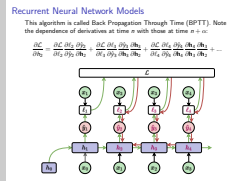


Recurrent Neural Network Models

This algorithm is called Back Propagation Through Time (BPTT). Note the dependence of derivatives at time n with those at time $n+\alpha$:

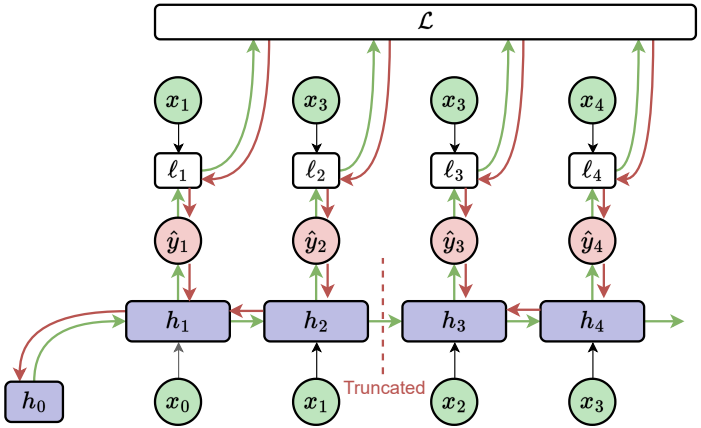
$$\frac{\partial \mathcal{L}}{\partial h_2} = \frac{\partial \mathcal{L}}{\partial \ell_2}\frac{\partial \ell_2}{\partial \hat{y}_2}\frac{\partial \hat{y}_2}{\partial \mathbf{h}_2} + \frac{\partial \mathcal{L}}{\partial \ell_3}\frac{\partial \ell_3}{\partial \hat{y}_3}\frac{\partial \hat{y}_3}{\partial \mathbf{h}_3}\frac{\partial \mathbf{h}_3}{\partial \mathbf{h}_2} + \frac{\partial \mathcal{L}}{\partial \ell_4}\frac{\partial \ell_4}{\partial \hat{y}_4}\frac{\partial \hat{y}_4}{\partial \mathbf{h}_4}\frac{\partial \mathbf{h}_4}{\partial \mathbf{h}_3}\frac{\partial \mathbf{h}_3}{\partial \mathbf{h}_2} + \dots$$

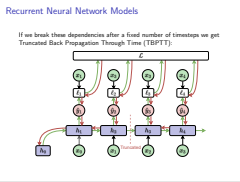




Recurrent Neural Network Models

If we break these dependencies after a fixed number of timesteps we get Truncated Back Propagation Through Time (TBPTT):



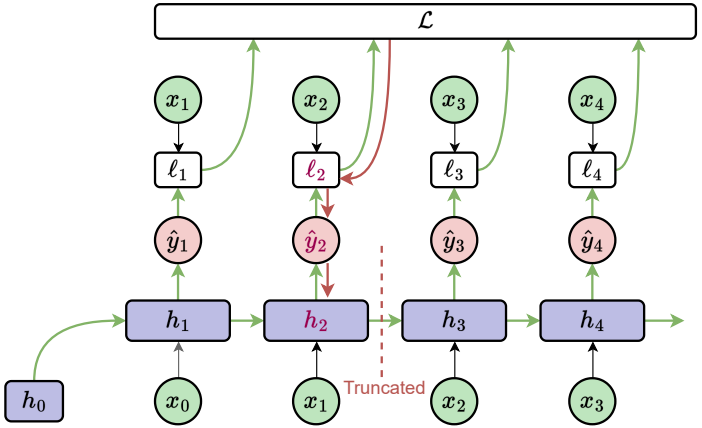


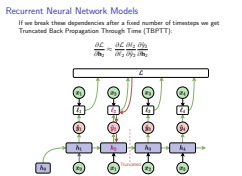
If our training data has many very long sequences, or in the limit it may consist of a single sequence, the BPTT algorithm can become impractical to apply, as the computation and memory required grows linearly with the length of the sequence (recall that for backpropagation we need to store the hidden states for every layer/timestep). If our data is one long sequence we also have to perform a forward and backward pass through all of it before we can do a single SGD step. In order to overcome these difficulties the truncated backpropagation through time algorithm (TBPTT) splits long sequences into fixed length sequences of length k. A forward and backward pass is then performed on each sub-sequence. Note that we can initialise the forward pass with the last hidden state from the previous sub-sequence, so the forward pass remains exact. However the back pass now discards gradient information across sub-sequences and thus is an approximation. Thus TBPTT has complexity linear in k, rather than the length of the longest training sequence. While approximate, the TBPTT algorithm is often effective for learning medium range dependencies in long sequences.

Recurrent Neural Network Models

If we break these dependencies after a fixed number of timesteps we get Truncated Back Propagation Through Time (TBPTT):

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}_2} \approx \frac{\partial \mathcal{L}}{\partial \ell_2} \frac{\partial \ell_2}{\partial \hat{y}_2} \frac{\partial \hat{y}_2}{\partial \mathbf{h}_2}$$



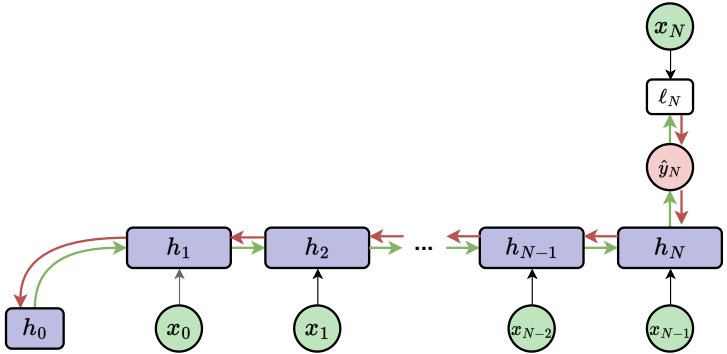


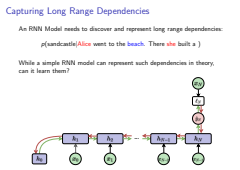
Capturing Long Range Dependencies

An RNN Model needs to discover and represent long range dependencies:

$p(\text{sandcastle}|\text{Alice went to the beach. There she built a })$

While a simple RNN model can represent such dependencies in theory, can it learn them?



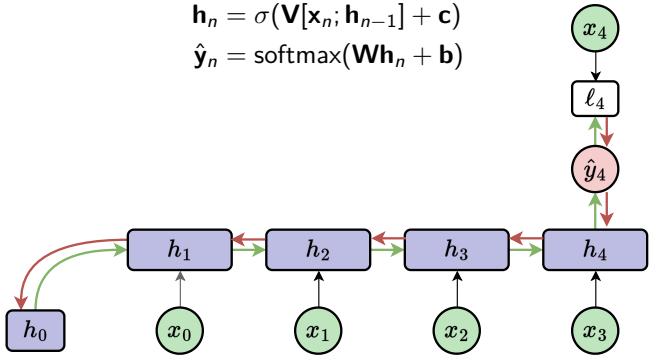


A key reason to use an RNN for sequence modelling is to capture long distance dependencies in the data, such as the example in the left. Earlier we saw that we can consider an unrolled RNN to be a deep feed forward network, but we have also seen how such networks can suffer from the vanishing gradient problem, where the repeated application of non-linearities can shrink the gradient. RNNs will suffer from the same problem, but they also involve the repeated application of the same transformation matrix $\mathbf{V}$.

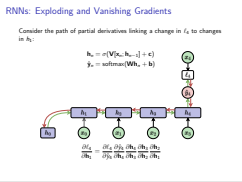
RNNs: Exploding and Vanishing Gradients

Consider the path of partial derivatives linking a change in ℓ_4 to changes in h_1:

$$\mathbf{h}_n = \sigma(\mathbf{V}[\mathbf{x}_n; \mathbf{h}_{n-1}] + \mathbf{c})$$
$$\hat{\mathbf{y}}_n = \text{softmax}(\mathbf{W}\mathbf{h}_n + \mathbf{b})$$



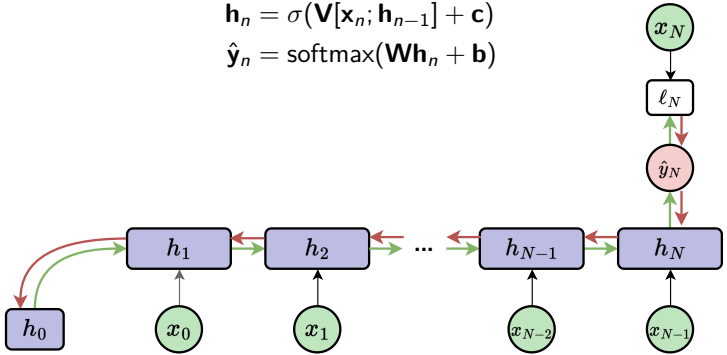
$$\frac{\partial \ell_4}{\partial \mathbf{h}_1} = \frac{\partial \ell_4}{\partial \hat{y}_4} \frac{\partial \hat{y}_4}{\partial \mathbf{h}_4} \frac{\partial \mathbf{h}_4}{\partial \mathbf{h}_3} \frac{\partial \mathbf{h}_3}{\partial \mathbf{h}_2} \frac{\partial \mathbf{h}_2}{\partial \mathbf{h}_1}$$



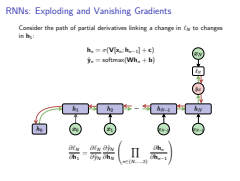
RNNs: Exploding and Vanishing Gradients

Consider the path of partial derivatives linking a change in ℓ_N to changes in $\mathbf{h}_1$:

$$\mathbf{h}_n = \sigma(\mathbf{V}[\mathbf{x}_n; \mathbf{h}_{n-1}] + \mathbf{c})$$
$$\hat{\mathbf{y}}_n = \text{softmax}(\mathbf{W}\mathbf{h}_n + \mathbf{b})$$



$$\frac{\partial \ell_N}{\partial \mathbf{h}_1} = \frac{\partial \ell_N}{\partial \hat{y}_N} \frac{\partial \hat{y}_N}{\partial \mathbf{h}_N} \left(\prod_{n \in \{N, \dots, 2\}} \frac{\partial \mathbf{h}_n}{\partial \mathbf{h}_{n-1}} \right)$$



We see that the core of this computation is the repeated product of partial derivatives of each hidden state with respect to the previous one.

RNNs: Exploding and Vanishing Gradients

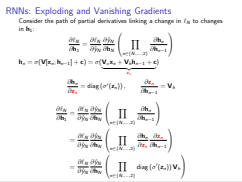
Consider the path of partial derivatives linking a change in ℓ_N to changes in $\mathbf{h}_1$:

$$\frac{\partial \ell_N}{\partial \mathbf{h}_1} = \frac{\partial \ell_N}{\partial \hat{y}_N} \frac{\partial \hat{y}_N}{\partial \mathbf{h}_N} \left(\prod_{n \in \{N,\dots,2\}} \frac{\partial \mathbf{h}_n}{\partial \mathbf{h}_{n-1}} \right)$$

$$\mathbf{h}_n = \sigma(\mathbf{V}[\mathbf{x}_n; \mathbf{h}_{n-1}] + \mathbf{c}) = \sigma(\underbrace{\mathbf{V}_x \mathbf{x}_n + \mathbf{V}_h \mathbf{h}_{n-1} + \mathbf{c}}_{\mathbf{z}_n})$$

$$\frac{\partial \mathbf{h}_n}{\partial \mathbf{z}_n} = \operatorname{diag}\left(\sigma'(\mathbf{z}_n)\right), \qquad \frac{\partial \mathbf{z}_n}{\partial \mathbf{h}_{n-1}} = \mathbf{V}_h$$

$$\begin{aligned}
\frac{\partial \ell_N}{\partial \mathbf{h}_1} &= \frac{\partial \ell_N}{\partial \hat{y}_N} \frac{\partial \hat{y}_N}{\partial \mathbf{h}_N} \left(\prod_{n \in \{N,\dots,2\}} \frac{\partial \mathbf{h}_n}{\partial \mathbf{h}_{n-1}} \right) \\
&= \frac{\partial \ell_N}{\partial \hat{y}_N} \frac{\partial \hat{y}_N}{\partial \mathbf{h}_N} \left(\prod_{n \in \{N,\dots,2\}} \frac{\partial \mathbf{h}_n}{\partial \mathbf{z}_n} \frac{\partial \mathbf{z}_n}{\partial \mathbf{h}_{n-1}} \right) \\
&= \frac{\partial \ell_N}{\partial \hat{y}_N} \frac{\partial \hat{y}_N}{\partial \mathbf{h}_N} \left(\prod_{n \in \{N,\dots,2\}} \operatorname{diag}\left(\sigma'(\mathbf{z}_n)\right) \mathbf{V}_h \right)
\end{aligned}$$



Writing the partial derivative in this way makes clear that the core of the recurrent product in the BPTT algorithm is the repeated multiplication of $\mathbf{V}_h$. When raising a matrix to a power we have three possible outcomes. If the largest eigenvalue of Vh is 1, then gradient will propagate, if it is ¿ 1, the product will grow exponentially (explode), and if it is ¡ 1, the product shrinks exponentially (vanishes). Most of the time the spectral radius of Vh is small. The result is that the gradient vanishes and long range dependencies are not learnt.

RNNs: Exploding and Vanishing Gradients

$$\frac{\partial \ell_N}{\partial \mathbf{h}_1} = \frac{\partial \ell_N}{\partial \hat{y}_N} \frac{\partial \hat{y}_N}{\partial \mathbf{h}_N} \left(\prod_{n \in \{N, \ldots, 2\}} \text{diag}\left(\sigma'(\mathbf{z}_n)\right) \mathbf{V}_h \right)$$

The core of the recurrent product is the repeated multiplication of $\mathbf{V}_h$. If the largest eigenvalue of $\mathbf{V}_h$ is

- 1, then gradient will propagate,
- > 1, the product will grow exponentially (explode),
- < 1, the product shrinks exponentially (vanishes).

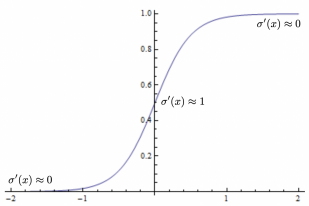
RNNs: Exploding and Vanishing Gradients
The core of the recurrent product is the repeated multiplication of $\mathbf{V}_h$. If the largest eigenvalue of $\mathbf{V}_h$ is
- 1, then gradient will propagate,
- > 1, the product will grow exponentially (explode),
- < 1, the product shrinks exponentially (vanishes).

When raising a matrix to a power we have three possible outcomes. If the largest eigenvalue of $\mathbf{V}_h$ is 1, then gradient will propagate, if it is > 1, the product will grow exponentially (explode), and if it is < 1, the product shrinks exponentially (vanishes).

RNNs: Exploding and Vanishing Gradients

Most of the time the spectral radius of $\mathbf{V}_h$ is small. The result is that the gradient vanishes and long range dependencies are not learnt.

Many non-linearities ($\sigma(\cdot)$) can also shrink the gradient.



Second order optimizers ((Quasi-)Newtonian Methods) can overcome this, but they are difficult to scale. Careful initialization of the recurrent weights can help[5].

The most popular solution to this issue is to change the network architecture to include gated units[6].

[5] Stephen Merity: Explaining and illustrating orthogonal initialization for recurrent neural networks. `https://smerity.com/articles/2016/orthogonal_init.html`

[6] Christopher Olah: Understanding LSTM Networks (`https://colah.github.io/posts/2015-08-Understanding-LSTMs/`)

RNNs: Exploding and Vanishing Gradients
Most of the time the spectral radius of $\mathbf{V}_h$ is small. The result is that the gradient vanishes and long range dependencies are not learnt.
Many non-linearities ($\sigma(\cdot)$) can also shrink the gradient.
Second order optimizers ((Quasi-)Newtonian Methods) can overcome this, but they are difficult to scale. Careful initialization of the recurrent weights can help[5].
The most popular solution to this issue is to change the network architecture to include gated units[6].
[5] Stephen Merity: Explaining and illustrating orthogonal initialization for recurrent neural networks. https://smerity.com/articles/2016/orthogonal_init.html
[6] Christopher Olah: Understanding LSTM Networks (https://colah.github.io/posts/2015-08-Understanding-LSTMs/)

Outline

Neural Networks

Training Deep Neural Networks

Convolution Neural Networks

Recurrent Neural Networks

Transformers

Outline

Neural Networks

Training Deep Neural Networks

Convolution Neural Networks

Recurrent Neural Networks

Transformers

Issues with recurrent models

Lack of parallelizability: Forward and backward passes have $\mathcal{O}$(sequence length) unparallelizable operations

- GPUs can perform a bunch of independent computations at once!
- But future RNN hidden states can't be computed in full before past RNN hidden states have been computed
- Inhibits training on very large datasets!

If not recurrence, then what? How about attention?

Attention treats each timestep as a query to access and incorporate information from a set of values.

Issues with recurrent models

Lack of parallelizability: Forward and backward passes have $\mathcal{O}$(sequence length) unparallelizable operations
- GPUs can perform a bunch of independent computations at once!
- But future RNN hidden states can't be computed in full before past RNN hidden states have been computed
- Inhibits training on very large datasets!

If not recurrence, then what? How about attention?

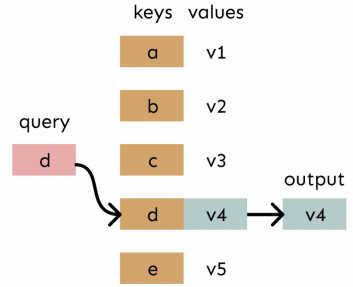
Attention treats each timestep as a query to access and incorporate information from a set of values.

Attention

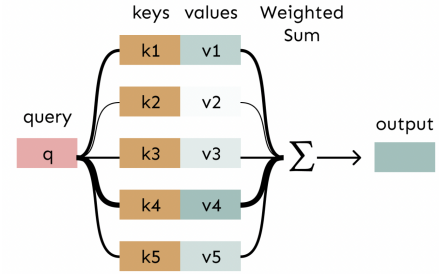
Attention as a soft, averaging lookup table

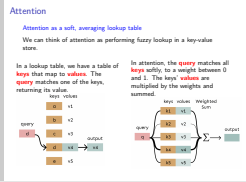
We can think of attention as performing fuzzy lookup in a key-value store.

In a lookup table, we have a table of **keys** that map to **values**. The **query** matches one of the keys, returning its value.



In attention, the **query** matches all **keys** softly, to a weight between 0 and 1. The keys' **values** are multiplied by the weights and summed.





Attention

The input consists of queries and keys of dimension d_k, and values of dimension d_v.

Compute the dot products of the query with all keys, divide each by $\sqrt{d_k}$, and apply a softmax function to obtain the weights on the values

Compute the attention function on a set of queries simultaneously, packed together into a matrix $\mathbf{Q}$

Keys and values are also packed together into matrices $\mathbf{K}$ and $\mathbf{V}$

Attention is computed as

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}$$

Attention

The input consists of queries and keys of dimension d_k, and values of dimension d_v.

Compute the dot products of the query with all keys, divide each by $\sqrt{d_k}$, and apply a softmax function to obtain the weights on the values

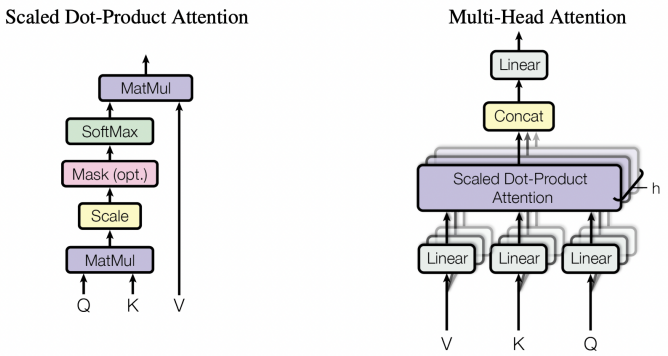
Compute the attention function on a set of queries simultaneously, packed together into a matrix $\mathbf{Q}$

Keys and values are also packed together into matrices $\mathbf{K}$ and $\mathbf{V}$

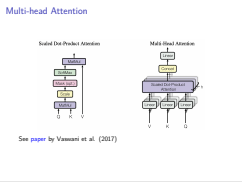
Attention is computed as

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}$$

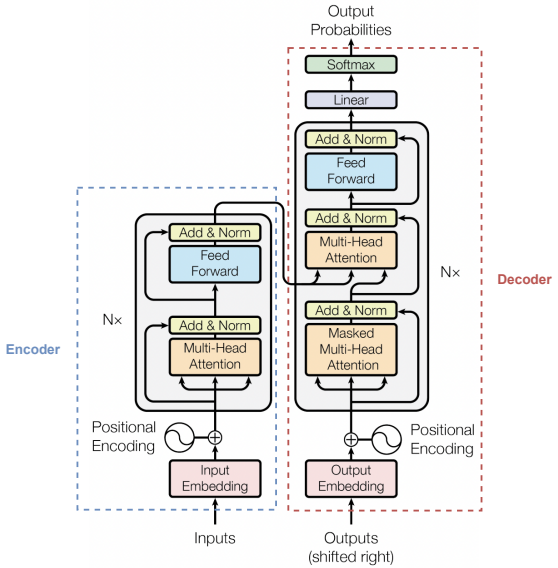
Multi-head Attention



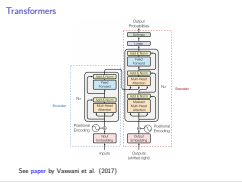
See paper by Vaswani et al. (2017)



Transformers



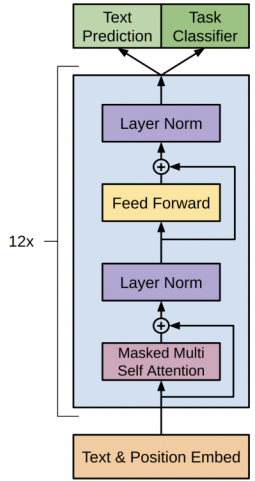
See paper by Vaswani et al. (2017)



GPT and BERT

Decoder part

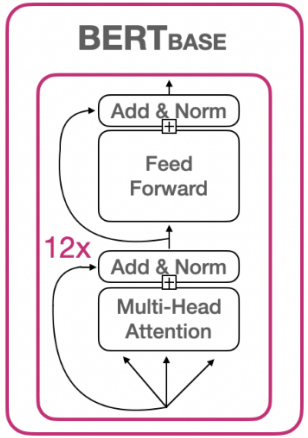
GPT: Generative Pre-trained Transformers



See paper by Radford et al. (2018)

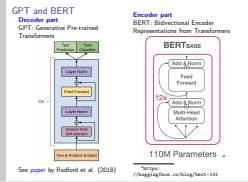
Encoder part

BERT: Bidirectional Encoder Representations from Transformers



110M Parameters [a]

[a] https://huggingface.co/blog/bert-101



Recap

Neural Networks

Training Deep Neural Networks

Convolution Neural Networks

Recurrent Neural Networks

Transformers

Recap
Neural Networks
Training Deep Neural Networks
Convolution Neural Networks
Recurrent Neural Networks
Transformers

END LECTURE

END LECTURE