Introduction to Modern Machine Learning Part 2: Deep Learning and Generative Modelling

Cristopher Salvi

Summer School on Machine Learning, Applied Statistics, and Quantitative Finance

Outline

1. Introduction to Deep Learning
   - Multi-layer perceptrons (MLPs)
   - Backpropagation
   - Gradient descent training and optimisers
   - Initialisation, overfitting and parallel training
2. Neural networks for images
   - Convolutions in 1d and 2d
   - Convolutional neural networks (CNNs)
   - Padding and pooling layers
   - Normalisation layers
3. Neural networks for sequences
   - Recurrent neural networks (RNNs)
   - Exploding and vanishing gradients
   - Gated recurrent unit (GRU) & Long-short term memory (LSTM)
   - Attention and Transformers
4. Generative modelling for text
   - Bidirectional Encoder Representations from Transformers (BERT)
   - Introduction to Large Language Models (LLMs)

- All example code can be found at https://github.com/crispitagorico/summer_school

1) Introduction to Deep Learning

Perceptron

- A perceptron is a mapping of the form:

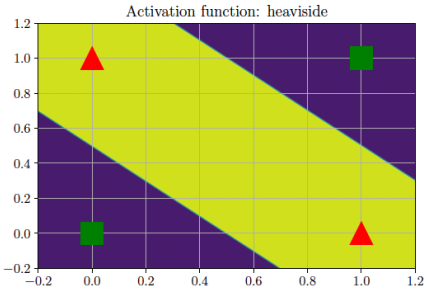
$$f(x;\theta) = \mathbb{I}(\omega^\top x + b \geq 0)$$

where $\mathbb{I}$ is the Heaviside step function.

- The XOR function $f_{\text{XOR}} : \{0,1\} \to \{0,1\}$ computes the exclusive OR of its two binary inputs. See truth table.

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

- This function is not linearly separable, so a single perceptron cannot represent it.



Multilayer perceptron (MLP)

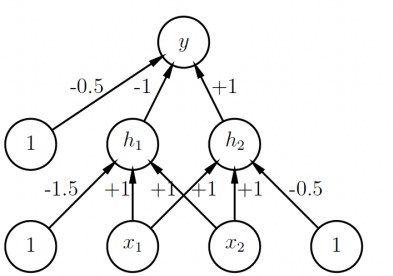
- To represent non-linear functions like XOR we can stack multiple perceptrons on top of each other. This is called a multilayer perceptron (MLP).
- For example, to solve the XOR problem, we can use the following MLP:

- The first hidden unit h_1 computes the AND operation so that h_1 will fire only if x_1 AND x_2 are both on since then

$$w_1^\top x + b_1 = [1,1]^\top [1,1] - 1.5 = 0.5 > 0.$$

- Similarly, the second hidden unit h_2 computes the OR operation, and the third computes the output given by NOT h_1 AND h_2.
- Thus, the network computes a function equivalent to XOR

$$y = f(x_1, x_2) = \overline{(x_1 \wedge x_2)} \wedge (x_1 \vee x_2).$$



Differentiable MLPs

- By generalizing this example, we can show that an MLP can represent any logical function.
- However, we obviously want to avoid having to specify the weights and biases by hand. In the rest of this section, we will discuss ways to learn these parameters from data.
- The Heaviside function is non-differentiable. This makes such models difficult to train.
- We replace the Heaviside function with a differentiable activation function $\varphi : \mathbb{R} \to \mathbb{R}$ and define the hidden units z_ℓ at each layer ℓ as

$$z_\ell = \varphi(W_\ell z_{\ell-1} + b_\ell), \quad \ell = 1, ..., L.$$

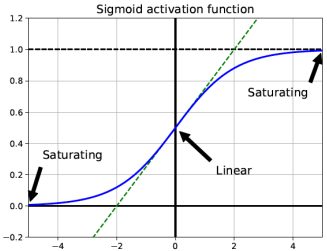
- This is equivalent to composing L of these functions to obtain a deep neural network (DNN)

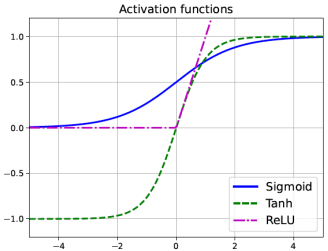
$$f(x) = f_L(f_{L-1}(...(f_1(x))...)), \quad f_\ell = \varphi(W_\ell x + b_\ell).$$

- We can compute the gradient of the output wrt the parameters (W_ℓ, b_ℓ for $\ell = 1, ..., L$) in each layer using the chain rule, also known as backpropagation. We can then pass the gradient to an optimizer, and thus minimize some training objective (more on this later).

Activation functions (1/2)

- Example of popular differentiable activation functions include Sigmoid and Tanh:



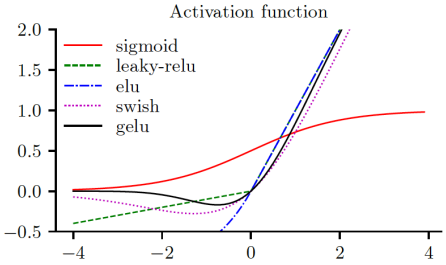


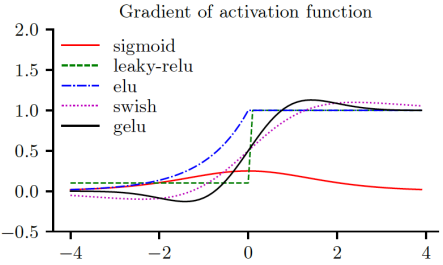
- However, both saturates at 1 for large positive inputs, and at 0 and -1 respectively for large negative inputs.
- In the saturated regimes, the gradient of the output wrt the input will be close to zero, so any gradient signal from higher layers will not be able to propagate back to earlier layers.
- This is called the vanishing gradient problem, and it makes it hard to train the model using gradient descent (more on this later).
- One of the keys to being able to train very deep models is to use non-saturating activation functions such as the rectified linear unit (ReLU)

$$\text{ReLU}(x) = \max(x, 0).$$

Activation functions (2/2)

Name	Definition	Range
Sigmoid	$\sigma(a) = \frac{1}{1+e^{-a}}$	$[0, 1]$
Hyperbolic tangent	$\tanh(a) = 2\sigma(2a) - 1$	$[-1, 1]$
Softplus	$\sigma_+(a) = \log(1 + e^a)$	$[0, \infty]$
Rectified linear unit	$\text{ReLU}(a) = \max(a, 0)$	$[0, \infty]$
Leaky ReLU	$\max(a, 0) + \alpha \min(a, 0)$	$[-\infty, \infty]$
Exponential linear unit	$\max(a, 0) + \min(\alpha(e^a - 1), 0)$	$[-\infty, \infty]$
Swish	$a\sigma(a)$	$[-\infty, \infty]$
GELU	$a\Phi(a)$	$[-\infty, \infty]$
Sine	$\sin(a)$	$[-1, 1]$





Backpropagation

- Consider a mapping of the form $\mathbf{o} = f(\mathbf{x})$, where $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{o} \in \mathbb{R}^m$. We assume that f is defined as a composition of functions:

$$f \;=\; f_4 \circ f_3 \circ f_2 \circ f_1,$$

  where $f_1 : \mathbb{R}^n \to \mathbb{R}^{m_1}$, $f_2 : \mathbb{R}^{m_1} \to \mathbb{R}^{m_2}$, $f_3 : \mathbb{R}^{m_2} \to \mathbb{R}^{m_3}$, and $f_4 : \mathbb{R}^{m_3} \to \mathbb{R}^m$.

- Define the intermediate steps needed to compute $\mathbf{o} = f(\mathbf{x})$: $\mathbf{x}_2 = f_1(\mathbf{x})$, $\mathbf{x}_3 = f_2(\mathbf{x}_2)$, $\mathbf{x}_4 = f_3(\mathbf{x}_3)$, and $\mathbf{o} = f_4(\mathbf{x}_4)$.

- Recall that for a differentiable function $f : \mathbb{R}^n \to \mathbb{R}^m$ the Jacobian is defined as:

$$\mathbf{J}_f(\mathbf{x}) = \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{pmatrix} = \begin{pmatrix} \nabla f_1(\mathbf{x})^\top \\ \vdots \\ \nabla f_m(\mathbf{x})^\top \end{pmatrix} = \left(\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n} \right) \in \mathbb{R}^{m \times n}$$

  where $\nabla f_i(\mathbf{x})^\top \in \mathbb{R}^{1 \times n}$ is the i'th row and $\frac{\partial f}{\partial x_j} \in \mathbb{R}^m$ is the j'th column.

- We can compute the Jacobian $\mathbf{J}_f(\mathbf{x}) = \frac{\partial \mathbf{o}}{\partial \mathbf{x}} \in \mathbb{R}^{m \times n}$ using the chain rule:

$$\frac{\partial \mathbf{o}}{\partial \mathbf{x}} = \frac{\partial \mathbf{o}}{\partial \mathbf{x}_4} \frac{\partial \mathbf{x}_4}{\partial \mathbf{x}_3} \frac{\partial \mathbf{x}_3}{\partial \mathbf{x}_2} \frac{\partial \mathbf{x}_2}{\partial \mathbf{x}} = \frac{\partial f_4(\mathbf{x}_4)}{\partial \mathbf{x}_4} \frac{\partial f_3(\mathbf{x}_3)}{\partial \mathbf{x}_3} \frac{\partial f_2(\mathbf{x}_2)}{\partial \mathbf{x}_2} \frac{\partial f_1(\mathbf{x})}{\partial \mathbf{x}} = \mathbf{J}_{f_4}(\mathbf{x}_4) \mathbf{J}_{f_3}(\mathbf{x}_3) \mathbf{J}_{f_2}(\mathbf{x}_2) \mathbf{J}_{f_1}(\mathbf{x}).$$

Forward vs reverse mode differentiation

- We now discuss how to compute the Jacobian $\mathbf{J}_f(\mathbf{x})$ efficiently.
- We can extract the i'th row from $\mathbf{J}_f(\mathbf{x})$ by using a vector Jacobian product (VJP) of the form $\mathbf{e}_i^\top \mathbf{J}_f(\mathbf{x})$, where $\mathbf{e}_i \in \mathbb{R}^m$ is the unit basis vector.
- Similarly, we can extract the j'th column from $\mathbf{J}_f(\mathbf{x})$ by using a Jacobian vector product (JVP) of the form $\mathbf{J}_f(\mathbf{x})\mathbf{e}_j$, where $\mathbf{e}_j \in \mathbb{R}^n$.
- This shows that the computation of $\mathbf{J}_f(\mathbf{x})$ reduces to either n JVPs or m VJPs.
- If $n < m$, it is more efficient to compute $\mathbf{J}_f(\mathbf{x})$ for each column $j = 1, ..., n$ by using JVPs in a right-to-left manner. The right multiplication with a column vector $\mathbf{v}$ is

$$\mathbf{J}_f(\mathbf{x})\,\mathbf{v} = \underbrace{\mathbf{J}_{f_4}(\mathbf{x}_4)}_{m\times m_3}\underbrace{\mathbf{J}_{f_3}(\mathbf{x}_3)}_{m_3\times m_2}\underbrace{\mathbf{J}_{f_2}(\mathbf{x}_2)}_{m_2\times m_1}\underbrace{\mathbf{J}_{f_1}(\mathbf{x})}_{m_1\times n}\underbrace{\mathbf{v}}_{n\times 1},$$

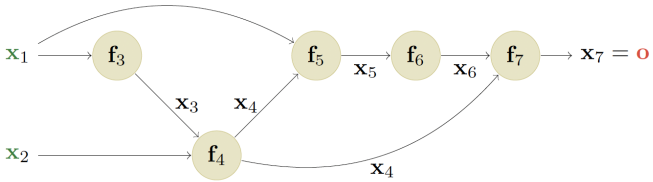
  This can be computed using forward mode differentiation.
- If $n > m$ (e.g. scalar output), it is more efficient to compute $\mathbf{J}_f(\mathbf{x})$ for each row $i = 1, ..., m$ by using VJPs in a left-to-right manner. The left multiplication with a row vector $\mathbf{u}^\top$ is

$$\mathbf{u}^\top \mathbf{J}_f(\mathbf{x}) = \underbrace{\mathbf{u}^\top}_{1\times m}\underbrace{\mathbf{J}_{f_4}(\mathbf{x}_4)}_{m\times m_3}\underbrace{\mathbf{J}_{f_3}(\mathbf{x}_3)}_{m_3\times m_2}\underbrace{\mathbf{J}_{f_2}(\mathbf{x}_2)}_{m_2\times m_1}\underbrace{\mathbf{J}_{f_1}(\mathbf{x})}_{m_1\times n},$$

  corresponding to the reverse-mode differentiation.

Computational graphs

- MLPs are a simple kind of DNN in which each layer feeds directly into the next, forming a densely connected graph.
- However, modern DNNs can combine differentiable components in much more complex ways, to create a more complex computation directed acyclic graph (DAG):



- The computational graph can be computed ahead of time, by using an API to define a static graph. This is how Tensorflow 1 worked.
- Alternatively, the graph can be computed just in time (JIT), by tracing the execution of the function on an input argument. This is how JAX and PyTorch work.
- The latter approach makes it easier to work with a dynamic graph, whose shape can change depending on the values computed by the function.

Gradient descent (GD)

- The standard approach to train DNNs is to minimise a loss function $\mathcal{L}(\theta)$ using backprop, computing gradients of $\mathcal{L}$ wrt θ and pass them to an off-the-shelf optimizer.
- This requires solving an optimisation problem, where we try to find the values for a set of variables $\theta \in \Theta$, that minimize a scalar-valued loss function $\mathcal{L}(\theta)$.
- Finding global optima is usually intractable (lack of convexity...). Thus, usually we will just try to find a local optimum.
- The most popular algorithm to do so is gradient descent (GD):

$$\theta_{t+1} = \theta_t - \eta_t \mathbf{g}_t, \quad \text{where } \mathbf{g}_t = \nabla_\theta \mathcal{L}(\theta)|_{\theta_t},$$

  and where η_t is called the learning rate (there are various ways of choosing it).
- If $\eta_t = \eta$ is constant, the learning rate can be found by finding the value that maximally decreases the objective along the chosen direction by solving the 1d minimization problem

$$\arg\min_{\eta > 0} \mathcal{L}(\theta_t - \eta \mathbf{g}_t)$$

  This method is known as line search.
- However, it is not usually necessary to be so precise, and η can be chosen heuristically.
- Gradient Descent Tutorial.

Convergence rates (1/2)

- For some simple problems, we can derive the convergence rate of GD explicitly. For example, consider a quadratic objective

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{2}\boldsymbol{\theta}^\top \mathbf{A}\boldsymbol{\theta} + \mathbf{b}^\top \boldsymbol{\theta} + c \quad \text{with} \quad \mathbf{A} \succ 0.$$

  Using GD with line search, one can show[1] that the convergence rate μ defined as

$$|\mathcal{L}(\boldsymbol{\theta}_{t+1}) - \mathcal{L}(\boldsymbol{\theta}_*)| \leq \mu\, |\mathcal{L}(\boldsymbol{\theta}_t) - \mathcal{L}(\boldsymbol{\theta}_*)|$$

  is given by

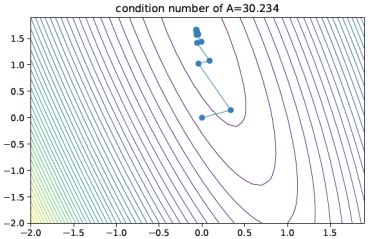
$$\mu = \left(\frac{\lambda_{\text{max}} - \lambda_{\text{min}}}{\lambda_{\text{max}} + \lambda_{\text{min}}} \right)^2$$

  where λ_{max} is the largest eigenvalue of $\mathbf{A}$ and λ_{min} is the smallest eigenvalue.
- We can rewrite this as $\mu = \left(\frac{\kappa-1}{\kappa+1}\right)^2$, where $\kappa = \frac{\lambda_{\text{max}}}{\lambda_{\text{min}}}$ is the condition number of $\mathbf{A}$.

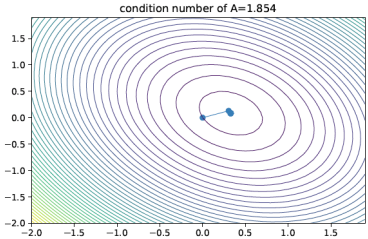
[1] D. Bertsekas. Convex Optimization Algorithms. Athena Scientific, 2015.

Convergence rates (2/2)

- We see that GD converges more quickly for the problem with smaller condition number.



(a)



(b)

- In the more general case of non-quadratic functions, the objective will often be locally quadratic around a local optimum. Hence the convergence rate depends on the condition number of the Hessian, $\kappa(\mathbf{H})$, at that point (more on this later).

Momentum

- Gradient descent can move very slowly along flat regions of the loss landscape.
- One simple solution, known as momentum method[2], is to move faster along directions that were previously good, and to slow down along directions where the gradient has suddenly changed, just like a ball rolling downhill.
- This can be implemented as follows:

$$\mathbf{m}_t = \beta \mathbf{m}_{t-1} + \mathbf{g}_{t-1}$$
$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \eta_t \mathbf{m}_t$$

  where $\mathbf{m}_t$ is the momentum and $0 < \beta < 1$. For $\beta = 0$, the method reduces to GD.
- We see that $\mathbf{m}_t$ is like an exponentially weighted moving average of the past gradients

$$\mathbf{m}_t = \beta \mathbf{m}_{t-1} + \mathbf{g}_{t-1} = \beta^2 \mathbf{m}_{t-2} + \beta \mathbf{g}_{t-2} + \mathbf{g}_{t-1} = \cdots = \sum_{\tau=0}^{t-1} \beta^\tau \mathbf{g}_{t-\tau-1}.$$

  We see that past gradients exhibit some cumulative influence on the present.
- More sophisticated momentum methods include Nesterov accelerated gradient[3].

[2] D. Bertsekas. Nonlinear Programming. Second. Athena Scientific, 1999.

[3] Y. Nesterov. Introductory Lectures on Convex Optimization. A basic course. Kluwer, 2004.

Second-order methods

- The above optimisation algorithms only use the gradient and are called first-order methods. They have the advantage that the gradient is cheap to compute and to store, but they do not model the curvature of the space, and hence they can be slower to converge.
- Second-order methods incorporate curvature in various ways (e.g., via the Hessian), which may yield faster convergence.
- The classic second-order method is Newton's method:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \mathbf{H}_t^{-1} \mathbf{g}_t$$

where the Hessian

$$\mathbf{H}_t \triangleq \nabla^2 \mathcal{L}(\boldsymbol{\theta})\big|_{\boldsymbol{\theta}_t} = \nabla^2 \mathcal{L}(\boldsymbol{\theta}_t) = \mathbf{H}(\boldsymbol{\theta}_t)$$

is assumed to be positive-definite (to ensure the update is well-defined).

- It can be derived by making a second-order Taylor approximation of $\mathcal{L}(\boldsymbol{\theta})$ around $\boldsymbol{\theta}_t$:

$$\mathcal{L}_{\text{quad}}(\boldsymbol{\theta}) = \mathcal{L}(\boldsymbol{\theta}_t) + \mathbf{g}_t^\top (\boldsymbol{\theta} - \boldsymbol{\theta}_t) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_t)^\top \mathbf{H}_t (\boldsymbol{\theta} - \boldsymbol{\theta}_t)$$

The minimum of $\mathcal{L}_{\text{quad}}$ is at

$$\boldsymbol{\theta} = \boldsymbol{\theta}_t - \mathbf{H}_t^{-1} \mathbf{g}_t$$

So if the quadratic approximation is a good one, we should pick $-\mathbf{H}_t \mathbf{g}_t$ as direction in GD.

- The intuition for why this is faster than gradient descent is that the matrix inverse $\mathbf{H}^{-1}$ "undoes" any skew in the local curvature (as in picture above).

Stochastic gradient descent (SGD)

- A popular class of stochastic optimisation algorithms minimises a "noisy" version of the loss function and is usually of the form:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_t, \mathbf{z}_t) = \boldsymbol{\theta}_t - \eta_t \mathbf{g}_t, \quad \text{where } \mathbf{z}_t \sim q.$$

  This method is known as stochastic gradient descent (SGD).

- For example, consider the empirical risk minimization loss

$$\mathcal{L}(\boldsymbol{\theta}_t) = \frac{1}{N} \sum_{n=1}^{N} \ell(y_n, f(\mathbf{x}_n; \boldsymbol{\theta}_t)) = \frac{1}{N} \sum_{n=1}^{N} \mathcal{L}_n(\boldsymbol{\theta}_t).$$

- The gradient of this objective has the form

$$\mathbf{g}_t = \frac{1}{N} \sum_{n=1}^{N} \nabla_{\boldsymbol{\theta}} \mathcal{L}_n(\boldsymbol{\theta}_t) = \frac{1}{N} \sum_{n=1}^{N} \nabla_{\boldsymbol{\theta}} \ell(y_n, f(\mathbf{x}_n; \boldsymbol{\theta}_t)).$$

  This requires summing over all N training examples, and thus can be slow if N is large.

- Fortunately, we can approximate this by sampling a minibatch of $B \ll N$ samples to get

$$\mathbf{g}_t \approx \frac{1}{|\mathcal{B}_t|} \sum_{n \in \mathcal{B}_t} \nabla_{\boldsymbol{\theta}} \mathcal{L}_n(\boldsymbol{\theta}_t) = \frac{1}{|\mathcal{B}_t|} \sum_{n \in \mathcal{B}_t} \nabla_{\boldsymbol{\theta}} \ell(y_n, f(\mathbf{x}_n; \boldsymbol{\theta}_t))$$

  where $\mathcal{B}_t$ is a set of randomly chosen examples to use at iteration t.

Preconditioned SGD

- Preconditioned SGD methods involve updates of the form $\boldsymbol{\theta}_t - \eta_t \mathbf{M}_t^{-1} \mathbf{g}_t$. Unfortunately the noise in the gradient estimates make it difficult to reliably estimate the Hessian, which makes it difficult to use second-order methods. In addition, it is expensive to solve for the update direction with a full preconditioning matrix.
- Therefore most practitioners use a diagonal preconditioners $\mathbf{M}_t^{-1}$. Popular examples include AdaGrad[4], AdaDelta[5], and Adam[6] (which combines AdaDelta with Momentum).

[4] J. Duchi, et al. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". JMLR 12 (2011), pp. 2121–2159.
[5] M. D. Zeiler. "ADADELTA: An Adaptive Learning Rate Method". In: (2012). arXiv: 1212.5701 [cs.LG].
[6] D. Kingma et al. "Adam: A Method for Stochastic Optimization". In: ICLR. 2015.

Exploding and vanishing gradients (1/2)

- When training very deep models, the gradient tends to become either very small (called the vanishing gradient problem) or very large (called the exploding gradient problem).
- The behaviour of the system depends on the eigenvectors of $\mathbf{J}$. Although $\mathbf{J}$ is a real-valued matrix, it is not (in general) symmetric, so its eigenvalues and eigenvectors can be complex-valued, with the imaginary components corresponding to oscillatory behaviour.
- Let λ be the spectral radius of $\mathbf{J}$, which is the maximum of the absolute values of the eigenvalues. If this is greater than 1, the gradient can explode; if this is less than 1, the gradient can vanish.

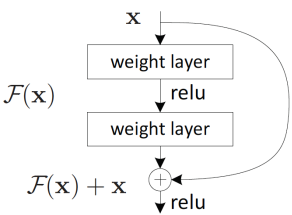
Exploding and vanishing gradients (2/2)

- The exploding gradient problem can be ameliorated by gradient clipping, in which we cap the magnitude of the gradient if it becomes too large, i.e., we use

$$\mathbf{g}' = \min\left(1, \frac{c}{\|\mathbf{g}\|}\right)\mathbf{g}$$

  This way, the norm of $\mathbf{g}'$ can never exceed c, but the vector is in the same direction as $\mathbf{g}$.
- The vanishing gradient problem is more difficult to solve. One can try different solutions:

- modifying the activation function (e.g. ReLU or leaky ReLU);
- initialise the DNN parameters in different ways (see below);
- incorporating residual connections (e.g. ResNet) so that gradients can flow directly from the output to earlier layers.
- Add extra normalization layers to the model, to normalise the statistics of the hidden units, e.g., to ensure they are zero mean and unit variance (more on this later).



Parameter initialisation

- Usually, parameters are initialised at random by sampling from a Normal distribution.
- Consider the function is given by $o_i = \sum_{j=1}^{n_{\text{in}}} w_{ij} x_j$; suppose $w_{ij} \sim \mathcal{N}(0, \sigma^2)$, and $\mathbb{E}[x_j] = 0$ and $\mathrm{V}[x_j] = \gamma^2$, where we assume x_j are independent of w_{ij}.
- The mean and variance of the output is given by

$$\mathbb{E}[o_i] = \sum_{j=1}^{n_{\text{in}}} \mathbb{E}[w_{ij} x_j] = \sum_{j=1}^{n_{\text{in}}} \mathbb{E}[w_{ij}]\mathbb{E}[x_j] = 0$$

$$\mathrm{V}[o_i] = \mathbb{E}[o_i^2] - (\mathbb{E}[o_i])^2 = \sum_{j=1}^{n_{\text{in}}} \mathbb{E}[w_{ij}^2 x_j^2] - 0 = \sum_{j=1}^{n_{\text{in}}} \mathbb{E}[w_{ij}^2]\mathbb{E}[x_j^2] = n_{\text{in}} \sigma^2 \gamma^2$$

- To keep the output variance from blowing up, we need to ensure $n_{\text{in}}\sigma^2 = 1$, where n_{in} is the fan-in of a unit (number of incoming connections).
- Analogously, in the backward pass, the variance of the gradients can blow up unless $n_{\text{out}}\sigma^2 = 1$, where n_{out} is the fan-out of a unit (number of outgoing connections).
- To satisfy both requirements at once, we set $\frac{1}{2}(n_{\text{in}} + n_{\text{out}})\sigma^2 = 1$, or equivalently

$$\sigma^2 = \frac{2}{n_{\text{in}} + n_{\text{out}}}$$

This is known as Xavier initialization or Glorot initialization.

Overfitting

- DNN are composed of multi million (or billion!) parameters. So overfitting is a serious issue.
- The simplest way to prevent overfitting in large DNN is called early stopping, the heuristic of stopping the training procedure when the error on the validation set starts to increase.
- Another approach is weight decay, where one imposes an ℓ_2 regularization of the objective.
- Another approach is dropout[7], where one randomly turns off all the outgoing connections from each neuron with probability p.

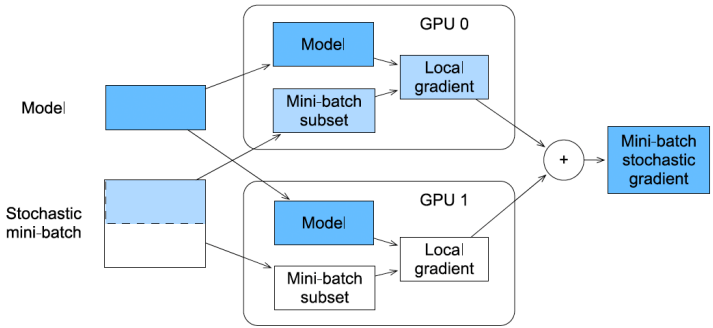
[7] N. Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: JMLR (2014).

Parallel training (1/2)

- It can be quite slow to train large models on large datasets. One way to speed this process up is to use specialized hardware, such as graphics processing units (GPUs) and tensor processing units (TPUs), which are very efficient at performing matrix-matrix multiplication.
- If we have multiple GPUs, we can sometimes further speed things up.
- There are two main approaches: model parallelism, in which we partition the model between machines, and data parallelism, in which each machine has its own copy of the model, and applies it to a different set of data.
- Model parallelism can be quite complicated, since it requires tight communication between machines to ensure they compute the correct answer. We will not discuss this further.

Parallel training (2/2)

- Data parallelism is generally much simpler: at each training step t, we do the following:
  1. we partition the minibatch across the K machines to get $\mathcal{D}_t^k$;
  2. each machine k computes its own gradient, $\mathbf{g}_t^k = \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}; \mathcal{D}_t^k)$;
  3. we collect all the local gradients on a central machine (e.g., device 0) and sum them using $\mathbf{g}_t = \sum_{k=1}^K \mathbf{g}_t^k$;
  4. we broadcast the summed gradient back to all devices, so $\tilde{\mathbf{g}}_t^k = \mathbf{g}_t$;
  5. each machine updates its own copy of the parameters using $\boldsymbol{\theta}_t^k := \boldsymbol{\theta}_t^k - \eta_t \tilde{\mathbf{g}}_t^k$.



2) Neural networks for images

Convolutions

- When the inputs of a DNN are images, we would like the model to be compatible with a variable-sized image as well as exhibiting translation-invariance.
- To solve these problems, we will use convolutional neural networks (CNNs), in which we replace matrix multiplication with a convolution operation.
- The convolution between two functions, say $f, g : \mathbb{R}^D \to \mathbb{R}$, is defined as

$$[f \circledast g](z) = \int_{\mathbb{R}^D} f(u)g(z-u)\,du.$$

- Now suppose we replace the functions with finite-length vectors, which we can think of as functions defined on a finite set of points. For example, suppose f is evaluated at the points $\{-L, -L+1, \ldots, -1, 0, 1, \ldots, L\}$ to yield the weight vector (also called a filter or kernel) $w_{-L} = f(-L)$ up to $w_L = f(L)$.
- Now let g be evaluated at points $\{-N, \ldots, N\}$ to yield the feature vector $x_{-N} = g(-N)$ up to $x_N = g(N)$. Then the above equation becomes

$$[\mathbf{w} \circledast \mathbf{x}](i) = w_{-L}x_{i+L} + \cdots + w_{-1}x_{i+1} + w_0 x_i + w_1 x_{i-1} + \cdots + w_L x_{i-L}.$$

- We see that we "flip" the weight vector $\mathbf{w}$ (since indices of $\mathbf{w}$ are reversed), and then "drag" it over the $\mathbf{x}$ vector, summing up the local windows at each point.

Convolutions in 1d

- E.g. the convolution of $\mathbf{x} = [1, 2, 3, 4]$ with $\mathbf{w} = [7, 6, 5]$ yields $\mathbf{z} = [5, 16, 34, 52, 45, 28]$.

-	-	1	2	3	4	-	-	
7	6	5	-	-	-	-	-	$z_0 = x_0 w_0 = 5$
-	7	6	5	-	-	-	-	$z_1 = x_0 w_1 + x_1 w_0 = 16$
-	-	7	6	5	-	-	-	$z_2 = x_0 w_2 + x_1 w_1 + x_2 w_0 = 34$
-	-	-	7	6	5	-	-	$z_3 = x_1 w_2 + x_2 w_1 + x_3 w_0 = 52$
-	-	-	-	7	6	5	-	$z_4 = x_2 w_2 + x_3 w_1 = 45$
-	-	-	-	-	7	6	5	$z_5 = x_3 w_2 = 28$

- There is a very closely related operation, in which we do not flip $\mathbf{w}$ first:

$$[\mathbf{w} * \mathbf{x}](i) = w_{-L} x_{i-L} + \cdots + w_{-1} x_{i-1} + w_0 x_i + w_1 x_{i+1} + \cdots + w_L x_{i+L}$$

  This is the convention used in the deep learning literature.

- We can also evaluate the weights $\mathbf{w}$ on domain $\{0, 1, \ldots, L-1\}$ and the features $\mathbf{x}$ on domain $\{0, 1, \ldots, N-1\}$, to eliminate negative indices. Then the above equation becomes

$$[\mathbf{w} \circledast \mathbf{x}](i) = \sum_{u=0}^{L-1} w_u x_{i+u}$$

Convolutions in 2d

- In 2d, convolutions become

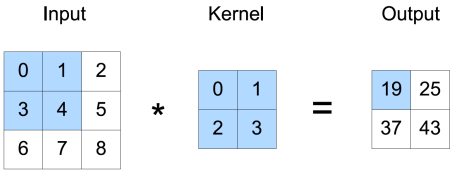
$$[\mathbf{W} \circledast \mathbf{X}](i,j) = \sum_{u=0}^{H-1} \sum_{v=0}^{W-1} w_{u,v} x_{i+u,j+v}$$

  where the 2d filter $\mathbf{W}$ has size $H \times W$.

- E.g. convolving a 3×3 input $\mathbf{X}$ with a 2×2 kernel $\mathbf{W}$ we get a 2×2 output $\mathbf{Y}$:

$$\mathbf{Y} = \begin{pmatrix} w_1 & w_2 \\ w_3 & w_4 \end{pmatrix} \circledast \begin{pmatrix} x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 \\ x_7 & x_8 & x_9 \end{pmatrix}$$

$$= \begin{pmatrix} w_1 x_1 + w_2 x_2 + w_3 x_4 + w_4 x_5 & w_1 x_2 + w_2 x_3 + w_3 x_5 + w_4 x_6 \\ w_1 x_4 + w_2 x_5 + w_3 x_7 + w_4 x_8 & w_1 x_5 + w_2 x_6 + w_3 x_8 + w_4 x_9 \end{pmatrix}$$



- In general, the input will have multiple channels (e.g., RGB images). We can extend the definition of convolution to this case by defining a kernel for each input channel.

Convolution as matmuls

- Since convolution is a linear operator, we can represent it by matrix multiplication by flattening the 2d matrix $\mathbf{X}$ into a 1d vector $\boldsymbol{x}$, and multiplying by a Toeplitz-like matrix $\mathbf{C}$ derived from the kernel $\mathbf{W}$, as follows:

$$\boldsymbol{y} = \mathbf{C}\boldsymbol{x} = \begin{pmatrix} w_1 & w_2 & 0 & w_3 & w_4 & 0 & 0 & 0 & 0 \\ 0 & w_1 & w_2 & 0 & w_3 & w_4 & 0 & 0 & 0 \\ 0 & 0 & 0 & w_1 & w_2 & 0 & w_3 & w_4 & 0 \\ 0 & 0 & 0 & 0 & w_1 & w_2 & 0 & w_3 & w_4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \end{pmatrix}$$

$$= \begin{pmatrix} w_1 x_1 + w_2 x_2 + w_3 x_4 + w_4 x_5 \\ w_1 x_2 + w_2 x_3 + w_3 x_5 + w_4 x_6 \\ w_1 x_4 + w_2 x_5 + w_3 x_7 + w_4 x_8 \\ w_1 x_5 + w_2 x_6 + w_3 x_8 + w_4 x_9 \end{pmatrix}$$

- We can recover the 2×2 output by reshaping the 4×1 vector $\boldsymbol{y}$ back to $\mathbf{Y}$.
- Thus we see that CNNs are like MLPs where the weight matrices have a sparse structure, and the elements are tied across spatial locations.
- This reduces the number of parameters compared to a weight matrix in a standard fully connected or dense layer, as used in MLPs.

Padding and striding (1/2)

- Convolving a 3×3 image with a 2×2 filter resulted in a 2×2 output. In general, convolving a $f_h \times f_w$ filter over an image of size $x_h \times x_w$ produces an output of size

$$(x_h - f_h + 1) \times (x_w - f_w + 1).$$

- If we want the output to have the same size as the input, we can use zero-padding, which means we add a border of 0s to the image.
- In general, if the input has size $x_h \times x_w$, we use a kernel of size $f_h \times f_w$, we use zero padding on each side of size p_h and p_w, then the output has size

$$(x_h + 2p_h - f_h + 1) \times (x_w + 2p_w - f_w + 1).$$

- If we set $2p = f - 1$, then the output will have the same size as the input.
- Since each output pixel is generated by a weighted combination of inputs in its receptive field, neighboring outputs will be very similar in value, since their inputs are overlapping.
- We can reduce this redundancy (and speedup computation) by skipping every s'th input. This is called strided convolution.
- In general, if the input has size $x_h \times x_w$, we use a kernel of size $f_h \times f_w$, we use zero padding on each side of size p_h and p_w, and we use strides of size s_h and s_w, then the output's size is:

$$\left\lfloor \frac{x_h + 2p_h - f_h + s_h}{s_h} \right\rfloor \times \left\lfloor \frac{x_w + 2p_w - f_w + s_w}{s_w} \right\rfloor$$

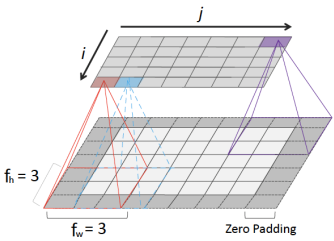
Padding and striding (2/2)

- In Figure (a) below we have $p = 1$, $f = 3$, $x_h = 5$ and $x_w = 7$, so the output has size

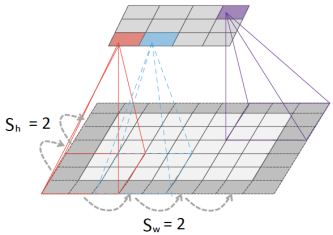
$$(5 + 2 - 3 + 1) \times (7 + 2 - 3 + 1) = 5 \times 7$$

- In Figure (b), we set the stride to $s = 2$ and the output (smaller than the input) has size

$$\left\lfloor \frac{5 + 2 - 3 + 2}{2} \right\rfloor \times \left\lfloor \frac{7 + 2 - 3 + 2}{2} \right\rfloor = \left\lfloor \frac{6}{2} \right\rfloor \times \left\lfloor \frac{4}{1} \right\rfloor = 3 \times 4$$



(a)



(b)

Pooling

- Convolution will preserve information about the location of input features (modulo reduced resolution), a property known as equivariance.
- In some case we want to be invariant to the location. For example, when performing image classification, we may just want to know if an object of interest (e.g., a face) is present anywhere in the image.
- One simple way to achieve this is called max pooling, which just computes the maximum over its incoming values.
- An alternative is to use average pooling, which replaces the max by the mean.

Normalisation layers

- The most popular normalization layer is called batch normalization (BN)[8].
- This ensures the distribution of the activations within a layer has zero mean and unit variance, when averaged across the samples in a minibatch.
- At test time, one usually use the frozen training values for the means and variances, rather than computing statistics from the test batch. Thus when using a model with BN, we need to specify if we are using it for inference or training.
- Batchnorm CNN Tutorial
- BN might struggle when the batch size is small, since the estimated mean and variance parameters can be unreliable. One solution is to compute the mean and variance by pooling statistics across other dimensions of the tensor (e.g. channel, height, width), but not across examples in the batch. This is known as layer normalization (LN)[9].

[8] S. Ioffe et al. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: ICML. 2015.

[9] J. L. Ba et al. "Layer Normalization". In: (2016). arXiv: 1607.06450.

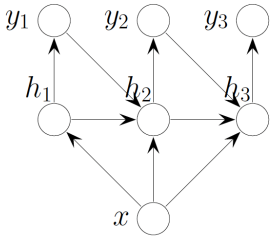
3) Neural networks for sequences

Recurrent neural networks (RNNs)

- Next, we discuss various kinds of neural networks for sequences. We will consider the case where the input is a sequence, the output is a sequence, or both are sequences.
- Such models have many applications, such as machine translation, speech recognition, text classification, text generation, image captioning, etc.
- A recurrent neural network (RNN) is a neural network which maps from an input space of sequences to an output space of sequences in a stateful way.
- That is, the prediction of output $\mathbf{y}_t$ depends not only on the input $\mathbf{x}_t$, but also on the hidden state of the system, $\mathbf{h}_t$, which gets updated over time, as the sequence is processed.
- Such models can be used for sequence generation, sequence classification, and sequence translation, as we explain next.

Vec2Seq (sequence generation) (1/2)

- RNNs can be used to learn functions mapping an input vector, to an output sequence. We call these vec2seq models.



- Let T be the (variable) length of the output sequence. The RNN then corresponds to the following conditional generative model:

$$p(\mathbf{y}_{1:T} \mid \mathbf{x}) = \sum_{\mathbf{h}_{1:T}} p(\mathbf{y}_{1:T}, \mathbf{h}_{1:T} \mid \mathbf{x}) = \sum_{\mathbf{h}_{1:T}} \prod_{t=1}^{T} p(\mathbf{y}_t \mid \mathbf{h}_t) p(\mathbf{h}_t \mid \mathbf{h}_{t-1}, \mathbf{y}_{t-1}, \mathbf{x})$$

where $\mathbf{h}_t$ is the hidden state, and where we define $p(\mathbf{h}_1 \mid \mathbf{h}_0, y_0, \mathbf{x}) = p(\mathbf{h}_1 \mid \mathbf{x})$ as the initial hidden state distribution (often deterministic).

- The output distribution is usually given by

$$p(\mathbf{y}_t \mid \mathbf{h}_t) = \mathrm{Cat}(\mathbf{y}_t \mid \mathrm{softmax}(\mathbf{W}_{hy}\mathbf{h}_t + \mathbf{b}_y))$$

where $\mathbf{W}_{hy}$ are the hidden-to-output weights, and $\mathbf{b}_y$ is the bias term.

Vec2Seq (sequence generation) (2/2)

- For real-valued outputs, we can use

$$p(\mathbf{y}_t \mid \mathbf{h}_t) = \mathcal{N}(y_t \mid \mathbf{W}_{hy}\mathbf{h}_t + \mathbf{b}_y, \sigma^2\mathbf{I})$$

- We assume the hidden state is computed deterministically as follows:

$$p(\mathbf{h}_t \mid \mathbf{h}_{t-1}, y_{t-1}, \mathbf{x}) = \mathbb{I}(\mathbf{h}_t = f(\mathbf{h}_{t-1}, y_{t-1}, \mathbf{x}))$$

  for some deterministic function f.

- The update function f is usually given by

$$\mathbf{h}_t = \varphi(\mathbf{W}_{xh}[\mathbf{x}; y_{t-1}] + \mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{b}_h)$$

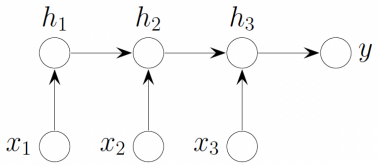
  where $\mathbf{W}_{hh}$ are the hidden-to-hidden weights, $\mathbf{W}_{xh}$ are the input-to-hidden weights, and $\mathbf{b}_h$ are the hidden biases.

- When we generate from an RNN, we sample from $\tilde{\mathbf{y}}_t \sim p(y_t \mid \mathbf{h}_t)$, and then "feed in" the sampled value into the hidden state, to deterministically compute $\mathbf{h}_{t+1} = f(\mathbf{h}_t, \tilde{\mathbf{y}}_t, \mathbf{x})$, from which we sample $\tilde{\mathbf{y}}_{t+1} \sim p(\mathbf{y}_{t+1} \mid \mathbf{h}_{t+1})$, etc.

- RNN Tutorial.

Seq2Vec (sequence classification) (1/2)

- Here we assume we have a single fixed-length output vector $\mathbf{y}$ we want to predict, given a variable length sequence as input. We call these seq2vec models. We focus on the case where the output is a class label $y \in \{1, ..., C\}$.



- The simplest approach is to use the final state of the RNN as input to the classifier:

$$p(y \mid \mathbf{x}_{1:T}) = \mathrm{Cat}(y \mid \mathrm{softmax}(\mathbf{W}\mathbf{h}_T)).$$

- We can often get better results if we let the hidden states of the RNN depend on the past and future context. To do this, we create two RNNs, one which recursively computes hidden states in the forwards direction, and one which recursively computes hidden states in the backwards direction. This is called a bidirectional RNN[10].

[10] M Schuster et al. "Bidirectional recurrent neural networks". In: IEEE. Trans on Signal Processing 45.11 (1997), pp. 2673– 2681.

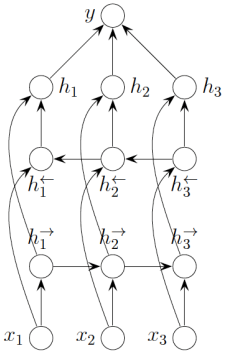
Seq2Vec (sequence classification) (2/2)

- More precisely, the model is defined as follows:

$$\vec{\mathbf{h}}_t = \varphi(\mathbf{W}^{\rightarrow}_{x\vec{h}}\mathbf{x}_t + \mathbf{W}^{\rightarrow}_{h\vec{h}}\vec{\mathbf{h}}_{t-1} + \mathbf{b}^{\rightarrow}_{\vec{h}})$$
$$\overleftarrow{\mathbf{h}}_t = \varphi(\mathbf{W}^{\leftarrow}_{x\overleftarrow{h}}\mathbf{x}_t + \mathbf{W}^{\leftarrow}_{h\overleftarrow{h}}\overleftarrow{\mathbf{h}}_{t+1} + \mathbf{b}^{\leftarrow}_{\overleftarrow{h}})$$

- We can then define $\mathbf{h}_t = [\vec{\mathbf{h}}_t, \overleftarrow{\mathbf{h}}_t]$ to be the representation of the state at time t, taking into account past and future information.
- Finally, we average pool over these hidden states to get the final classifier:

$$p(y \mid \mathbf{x}_{1:T}) = \mathrm{Cat}(y \mid \mathrm{softmax}(\mathbf{W}\bar{\mathbf{h}})), \quad \text{where } \bar{\mathbf{h}} = \frac{1}{T}\sum_{t=1}^{T}\mathbf{h}_t$$



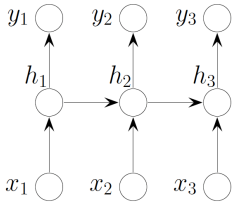
- Sentiment Analysis Tutorial.

Seq2Seq (sequence translation) (1/2)

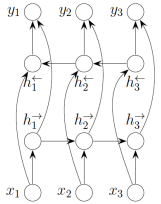
- Here we consider learning functions mapping sequences to sequences. These models are known as seq2seq. We distinguish between two cases: when the two sequences are of the same length and when they are not.
- When input and output sequences are of length T, it is straightforward to modify an RNN to solve this task (a):

$$p(\mathbf{y}_{1:T} \mid \mathbf{x}_{1:T}) = \sum_{\mathbf{h}_{1:T}} \prod_{t=1}^{T} p(y_t \mid \mathbf{h}_t) \mathbb{I}(\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{x}_t))$$

- We can often get better results if we let the decoder look into the "future" of $\mathbf{x}$ as well as the past, by using a bidirectional RNN (b).



(a)



(b)

Seq2Seq (sequence translation) (2/2)

- In the unaligned case ($T \neq T'$), one usually encodes the input sequence into a context vector $\mathbf{c} = f_{encoder}(\mathbf{x}_{1:T})$ using a Seq2Vec RNN encoder, and then generate the output sequence $\mathbf{y}_{1:T'} = f_{decoder}(\mathbf{c})$ using a Vec2Seq RNN encoder.
- When training a language model (more on this later), the likelihood of a sequence of words $w_1, w_2, \ldots, w_T$, is given by

$$p(\mathbf{w}_{1:T}) = \prod_{t=1}^{T} p(w_t \mid \mathbf{w}_{1:t-1})$$

- In an RNN, we therefore set the input to $x_t = w_{t-1}$ and the output to $y_t = w_t$. Note that we condition on the *ground truth* labels from the past, $\mathbf{w}_{1:t-1}$, not labels generated from the model. This is called teacher forcing, since the teacher's values are "force fed" into the model as input at each step (i.e., x_t is set to w_{t-1}).
- Unfortunately, teacher forcing can sometimes result in models that perform poorly at test time as the model has only ever been trained on inputs that are "correct", so it may not know what to do if, at test time, it encounters an input sequence $\mathbf{w}_{1:t-1}$ generated from the previous step that deviates from what it saw in training.
- A common solution starts off using teacher forcing, but at random time steps, feeds in samples from the model instead; the fraction of time this happens is gradually increased.

Backpropagation through time (1/3)

- We can compute the maximum likelihood estimate of the parameters for an RNN by solving

$$\theta^* = \arg\max_\theta p(\mathbf{y}_{1:T} \mid \mathbf{x}_{1:T}, \theta),$$

  where we have assumed a single training sequence for notational simplicity. To compute the MLE, we have to compute gradients of the loss with respect to the parameters.
- To do this, we can unroll the computation graph, and then apply the backpropagation algorithm. This is called backpropagation through time (BTT).
- More precisely, consider the following model:

$$\mathbf{h}_t = \mathbf{W}_{hx}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1}, \quad \mathbf{o}_t = \mathbf{W}_{ho}\mathbf{h}_t$$

  where $\mathbf{o}_t$ are the output logits, and where we drop the bias terms for notational simplicity.
- We define the loss to be $L = \frac{1}{T}\sum_{t=1}^{T} \ell(y_t, \mathbf{o}_t)$. We need to compute the derivatives

$$\frac{\partial L}{\partial \mathbf{W}_{hx}}, \quad \frac{\partial L}{\partial \mathbf{W}_{hh}}, \quad \text{and} \quad \frac{\partial L}{\partial \mathbf{W}_{ho}}.$$

  The latter term is easy, since it is local to each time step. However, the first two terms depend on the hidden state, and thus require working backwards in time.

Backpropagation through time (2/3)

- We simplify the notation by defining

$$\mathbf{h}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1}, \mathbf{w}_h), \quad \mathbf{o}_t = g(\mathbf{h}_t, \mathbf{w}_o)$$

  where $\mathbf{w}_h$ is the flattened version of $\mathbf{W}_{hh}$ and $\mathbf{W}_{hx}$ stacked together.
- We focus on computing $\frac{\partial L}{\partial \mathbf{w}_h}$. By the chain rule, we have

$$\frac{\partial L}{\partial \mathbf{w}_h} = \frac{1}{T}\sum_{t=1}^{T} \frac{\partial \ell(y_t, \mathbf{o}_t)}{\partial \mathbf{w}_h} = \frac{1}{T}\sum_{t=1}^{T} \frac{\partial \ell(y_t, \mathbf{o}_t)}{\partial \mathbf{o}_t} \frac{\partial g(\mathbf{h}_t, \mathbf{w}_o)}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{w}_h}$$

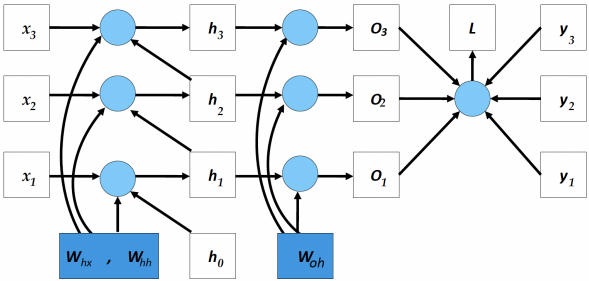
- We can expand the last term as follows:

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{w}_h} = \frac{\partial f(\mathbf{x}_t, \mathbf{h}_{t-1}, \mathbf{w}_h)}{\partial \mathbf{w}_h} + \frac{\partial f(\mathbf{x}_t, \mathbf{h}_{t-1}, \mathbf{w}_h)}{\partial \mathbf{h}_{t-1}} \frac{\partial \mathbf{h}_{t-1}}{\partial \mathbf{w}_h}$$

- If we expand this recursively, we find the following result:

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{w}_h} = \frac{\partial f(\mathbf{x}_t, \mathbf{h}_{t-1}, \mathbf{w}_h)}{\partial \mathbf{w}_h} + \sum_{i=1}^{t-1} \left(\prod_{j=i+1}^{t} \frac{\partial f(\mathbf{x}_j, \mathbf{h}_{j-1}, \mathbf{w}_h)}{\partial \mathbf{h}_{j-1}} \right) \frac{\partial f(\mathbf{x}_i, \mathbf{h}_{i-1}, \mathbf{w}_h)}{\partial \mathbf{w}_h}$$

Backpropagation through time (3/3)



- Unfortunately, this takes $O(T)$ time to compute per time step, for a total of $O(T^2)$ overall. It is therefore standard to truncate the sum to the most recent K terms.
- When using truncated BPTT, we can train the model with batches of short sequences, usually created by extracting non-overlapping subsequences from the original sequence. If the previous subsequence ends at time $t-1$, and the current subsequence starts at time t, we can "carry over" the hidden state of the RNN across batch updates during training. However, if the subsequences are not ordered, we need to reset the hidden state.

Vanishing and exploding gradients

- Unfortunately, the activations in an RNN can decay or explode as we go forwards in time, since we multiply by the weight matrix $\mathbf{W}_{hh}$ at each time step.
- Similarly, the gradients in an RNN can decay or explode as we go backwards in time, since we multiply the Jacobians at each time step.
- A simple heuristic is to use gradient clipping.
- More sophisticated methods attempt to control the spectral radius λ of the forward mapping, $\mathbf{W}_{hh}$, as well as the backwards mapping, given by the Jacobian $\mathbf{J}_{hh}$.
- The simplest way to control the spectral radius is to randomly initialize $\mathbf{W}_{hh}$ in such a way as to ensure $\lambda \approx 1$, and then keep it fixed (i.e., we do not learn $\mathbf{W}_{hh}$).
- In this case, only the output matrix $\mathbf{W}_{ho}$ needs to be learned, resulting in a convex optimization problem.
- This set of techniques is usually referred to as reservoir computing[11].
- An alternative is to modify the RNN architecture itself, to use additive rather than multiplicative updates to the hidden states, as we discuss next.

[11] H. Lukosevicius et al. "Reservoir computing approaches to RNN training". In: Computer Science Review 3.3 (2009), 127–149.

Gated recurrent units (1/2)

- Gated recurrent units (GRU)[12] learn when to update the hidden state, by using a gating unit to selectively "remember" important pieces of information, and learn when to reset the hidden state, and thus "forget" things that are no longer useful.
- We assume $\mathbf{X}_t$ is a $N \times D$ matrix, where N is the batch size, and D is the vocabulary size. Similarly, $\mathbf{H}_t$ is a $N \times H$ matrix, where H is the number of hidden units at time t.
- The reset gate $\mathbf{R}_t \in \mathbb{R}^{N \times H}$ and update gate $\mathbf{Z}_t \in \mathbb{R}^{N \times H}$ are computed using

$$\mathbf{R}_t = \text{Sigmoid}(\mathbf{X}_t \mathbf{W}_{xr} + \mathbf{H}_{t-1} \mathbf{W}_{hr} + \mathbf{b}_r), \quad \mathbf{Z}_t = \text{Sigmoid}(\mathbf{X}_t \mathbf{W}_{xz} + \mathbf{H}_{t-1} \mathbf{W}_{hz} + \mathbf{b}_z)$$

- Given this, we define a "candidate" next state vector using

$$\tilde{\mathbf{H}}_t = \tanh(\mathbf{X}_t \mathbf{W}_{xh} + (\mathbf{R}_t \odot \mathbf{H}_{t-1}) \mathbf{W}_{hh} + \mathbf{b}_h)$$

- This combines the old memories that are not reset (computed using $\mathbf{R}_t \odot \mathbf{H}_{t-1}$) with the new inputs $\mathbf{X}_t$. If the entries of the reset gate $\mathbf{R}_t$ are close to 1, we recover the standard RNN update rule. If the entries are close to 0, the model acts more like an MLP applied to $\mathbf{X}_t$. Thus the reset gate can capture new short-term information.

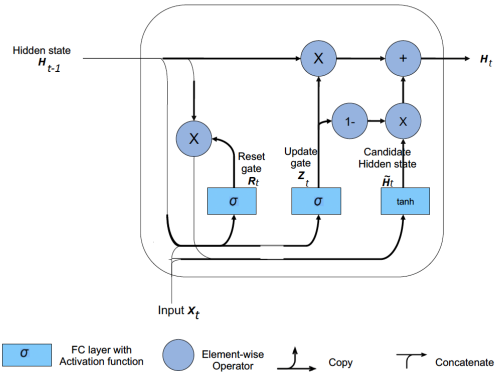
[12] K. Cho et al. "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation". In: EMNLP. 2014.

Gated recurrent units (GRU) (2/2)

- Once we have computed the candidate new state, the model computes the actual new state by using the dimensions from the candidate state $\tilde{\mathbf{H}}_t$ chosen by the update gate, $1 - \mathbf{Z}_t$, and keeping the remaining dimensions at their old values of $\mathbf{H}_{t-1}$:

$$\mathbf{H}_t = \mathbf{Z}_t \odot \mathbf{H}_{t-1} + (1 - \mathbf{Z}_t) \odot \tilde{\mathbf{H}}_t$$

- When $Z_{td} = 1$, we pass $H_{t-1,d}$ through unchanged, and ignore $\mathbf{X}_t$. Thus the update gate can capture long-term dependencies.



Long-short term memory (LSTM) (1/2)

- A more sophisticated version of the GRU is the long-short term memory (LSTM)[13]. The basic idea is to augment the hidden state $\mathbf{h}_t$ with a memory cell $\mathbf{c}_t$.
- We need three gates to control this cell: the output gate $\mathbf{O}_t$ determines what gets read out; the input gate $\mathbf{I}_t$ determines what gets read in; and the forget gate $\mathbf{F}_t$ determines when we should reset the cell.
- These gates are computed as follows:

$$\begin{aligned}\mathbf{O}_t &= \text{Sigmoid}(\mathbf{X}_t\mathbf{W}_{xo} + \mathbf{H}_{t-1}\mathbf{W}_{ho} + \mathbf{b}_o)\\ \mathbf{I}_t &= \text{Sigmoid}(\mathbf{X}_t\mathbf{W}_{xi} + \mathbf{H}_{t-1}\mathbf{W}_{hi} + \mathbf{b}_i)\\ \mathbf{F}_t &= \text{Sigmoid}(\mathbf{X}_t\mathbf{W}_{xf} + \mathbf{H}_{t-1}\mathbf{W}_{hf} + \mathbf{b}_f)\end{aligned}$$

- We then compute a candidate cell state:

$$\tilde{\mathbf{C}}_t = \tanh(\mathbf{X}_t\mathbf{W}_{xc} + \mathbf{H}_{t-1}\mathbf{W}_{hc} + \mathbf{b}_c)$$

- The actual update to the cell is either the candidate cell (if the input gate is on) or the old cell (if the not-forget gate is on):

$$\mathbf{C}_t = \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \tilde{\mathbf{C}}_t$$

- If $\mathbf{F}_t = 1$ and $\mathbf{I}_t = 0$, this can remember long-term memories.

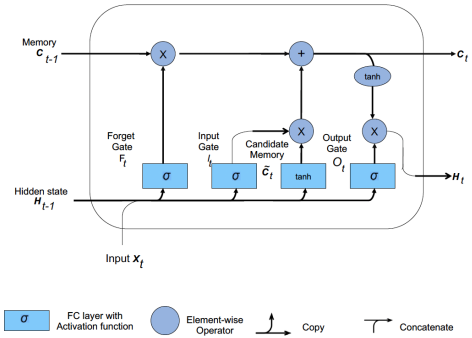
[13] S. Hochreiter et al. "Long short-term memory". In: Neural Computation 9.8 (1997), 1735–1780.

Long-short term memory (LSTM) (2/2)

- Finally, we compute the hidden state provided the output gate is on:

$$\mathbf{H}_t = \mathbf{O}_t \odot \tanh(\mathbf{C}_t)$$

- Note that $\mathbf{H}_t$ is used as the output of the unit as well as the hidden state for the next time step. This lets the model remember what it has just output (short-term memory), whereas the cell $\mathbf{C}_t$ acts as a long-term memory.
- LSTM Tutorial.



Attention

- In all of the neural networks we have considered so far, the hidden activations are a linear combination of the input activations, followed by a nonlinearity:

$$\mathbf{Z} = \varphi(\mathbf{X}\mathbf{W}),$$

where $\mathbf{X} \in \mathbb{R}^{m \times v}$ are the hidden feature vectors, and $\mathbf{W} \in \mathbb{R}^{v \times v'}$ are a fixed set of weights that are learned on a training set to produce $\mathbf{Z} \in \mathbb{R}^{m \times v'}$ outputs.

- However, we can use a more flexible model in which the weights depend on the inputs, i.e.,

$$\mathbf{Z} = \varphi(\mathbf{X}\mathbf{W}(\mathbf{X})).$$

- This kind of multiplicative interaction is called attention. More generally, we can write

$$\mathbf{Z} = \varphi(\mathbf{V}\mathbf{W}(\mathbf{Q}, \mathbf{K})),$$

where $\mathbf{Q} = \mathbf{Q}(\mathbf{X}) \in \mathbb{R}^{m \times q}$ are queries describing what each input is "looking for",
$\mathbf{K} = \mathbf{K}(\mathbf{X}) \in \mathbb{R}^{m \times q}$ are keys describing what each input vector contains, and
$\mathbf{V} = \mathbf{V}(\mathbf{X}) \in \mathbb{R}^{m \times v}$ are values describing how each input is transmitted to the output.

- We usually compute these quantities using linear projections of the input,

$$\mathbf{Q} = \mathbf{W}_q\mathbf{X}, \quad \mathbf{K} = \mathbf{W}_k\mathbf{X}, \quad \mathbf{V} = \mathbf{W}_v\mathbf{X}.$$

- When using attention to compute output z_j, we use its corresponding query q_j and compare it to each key k_i to get a similarity score, $0 \leq \alpha_{ij} \leq 1$, where $\sum_i \alpha_{ij} = 1$; we then set

$$z_j = \sum_i \alpha_{ij} v_i.$$

Softmax attention (1/2)

- We can think of attention as a dictionary lookup, in which we compare the query $\mathbf{q}$ to each key $\mathbf{k}_i$, and then retrieve the corresponding value $\mathbf{v}_i$.
- To make this lookup operation differentiable, instead of retrieving a single value $\mathbf{v}_i$, we compute a convex combination of the values, as follows:

$$\mathrm{Attn}(\mathbf{q}, (\mathbf{k}_1, \mathbf{v}_1), \ldots, (\mathbf{k}_m, \mathbf{v}_m)) = \mathrm{Attn}(\mathbf{q}, (\mathbf{k}_{1:m}, \mathbf{v}_{1:m})) = \sum_{i=1}^{m} \alpha_i(\mathbf{q}, \mathbf{k}_{1:m})\mathbf{v}_i \in \mathbb{R}^v$$

  where $\alpha_i(\mathbf{q}, \mathbf{k}_{1:m})$ is the i-th attention weight; again, these weights satisfy

$$0 \leq \alpha_i(\mathbf{q}, \mathbf{k}_{1:m}) \leq 1 \quad \text{and} \quad \sum_i \alpha_i(\mathbf{q}, \mathbf{k}_{1:m}) = 1.$$

- The attention weights can be computed from an attention score function $a(\mathbf{q}, \mathbf{k}_i) \in \mathbb{R}$, that computes the similarity of query $\mathbf{q}$ to key $\mathbf{k}_i$. Given the scores, we can compute the attention weights using the softmax function:

$$\alpha_i(\mathbf{q}, \mathbf{k}_{1:m}) = \mathrm{softmax}_i([a(\mathbf{q}, \mathbf{k}_1), \ldots, a(\mathbf{q}, \mathbf{k}_m)]) = \frac{\exp(a(\mathbf{q}, \mathbf{k}_i))}{\sum_{j=1}^{m} \exp(a(\mathbf{q}, \mathbf{k}_j))}$$

- It is standard practice to use scaled dot-product attention:

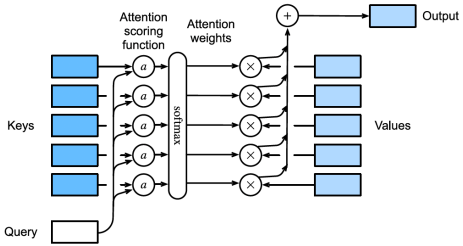
$$a(\mathbf{q}, \mathbf{k}) = \frac{\mathbf{q}^\top \mathbf{k}}{\sqrt{d}} \in \mathbb{R}$$

Softmax attention (2/2)

- In practice, we usually deal with minibatches of n vectors at a time. Let the corresponding matrices of queries, keys and values be denoted by $\mathbf{Q} \in \mathbb{R}^{n \times d}$, $\mathbf{K} \in \mathbb{R}^{m \times d}$, $\mathbf{V} \in \mathbb{R}^{m \times v}$. Then we can compute the attention-weighted outputs as follows:

$$\mathrm{Attn}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \mathrm{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}}\right)\mathbf{V} \in \mathbb{R}^{n \times v}$$

where the softmax function softmax is applied row-wise. Attention Tutorial.



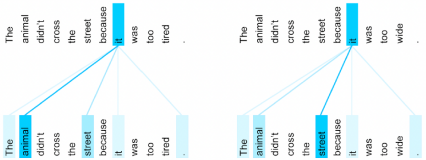
- In some cases, we might want to restrict attention to a subset of the dictionary, corresponding to valid entries. For example, we might want to pad sequences to a fixed length (for efficient minibatching), in which case we should "mask out" the padded locations, or use only the past to predict the future, in which case we should "mask out" the future locations. This is called masked attention.

Transformers

- The transformer model[14] is a seq2seq model which uses attention in the encoder as well as the decoder, thus eliminating the need for RNNs.
- Rather than the decoder attending to the encoder, we can modify the model so the encoder attends to itself. This is called self-attention.
- Given a sequence of input tokens $\mathbf{x}_1, \ldots, \mathbf{x}_n$, where $\mathbf{x}_i \in \mathbb{R}^d$, self-attention can generate a sequence of outputs of the same size using

$$\mathbf{y}_i = \mathrm{Attn}(\mathbf{x}_i, (\mathbf{x}_1, \mathbf{x}_1), \ldots, (\mathbf{x}_n, \mathbf{x}_n))$$

  where the query is $\mathbf{x}_i$, and the keys and values are all the inputs $\mathbf{x}_1, ..., \mathbf{x}_n$.
- To use this in a decoder, we can set $\mathbf{x}_i = \mathbf{y}_{i-1}$, and $n = i - 1$, so all the previously generated outputs are available.
- At training time, all the outputs are already known, so we can evaluate the above function in parallel, overcoming the sequential bottleneck of using RNNs.



[14] A. Vaswani et al. "Attention Is All You Need". In: NIPS. 2017.

Multi-head attention

- It is natural to want to use multiple attention matrices, to capture different notions of similarity. This is the basic idea behind multi-head attention.
- Given a query $\mathbf{q} \in \mathbb{R}^{d_q}$, keys $\mathbf{k}_j \in \mathbb{R}^{d_k}$, and values $\mathbf{v}_j \in \mathbb{R}^{d_v}$, we define the i'th attention head to be
$$\mathbf{h}_i = \text{Attn}(\mathbf{W}_i^{(q)}\mathbf{q}, \{\mathbf{W}_i^{(k)}\mathbf{k}_j, \mathbf{W}_i^{(v)}\mathbf{v}_j\}) \in \mathbb{R}^{p_v}$$
where $\mathbf{W}_i^{(q)} \in \mathbb{R}^{p_q \times d_q}$, $\mathbf{W}_i^{(k)} \in \mathbb{R}^{p_k \times d_k}$, and $\mathbf{W}_i^{(v)} \in \mathbb{R}^{p_v \times d_v}$ are projection matrices.
- We then stack the h heads together, and project to $\mathbb{R}^{p_o}$ using
$$\mathbf{h} = \text{MHA}(\mathbf{q}, \{\mathbf{k}_j, \mathbf{v}_j\}) = \mathbf{W}_o \begin{pmatrix} \mathbf{h}_1 \\ \vdots \\ \mathbf{h}_h \end{pmatrix} \in \mathbb{R}^{p_o}$$
where $\mathbf{W}_o \in \mathbb{R}^{p_o \times hp_v}$.
- If we set $p_q h = p_k h = p_v h = p_o$, we can compute all the output heads in parallel.
- Multi-Head Attention Tutorial.

Positional encoding (1/3)

- The performance of "vanilla" self-attention can be low, since attention is permutation invariant, and hence ignores the input word ordering.
- To overcome this, we can concatenate the word embeddings with a positional encoding, so that the model knows what order the words occur in.
- One way to do this is to represent each position by an integer. However, neural networks cannot natively handle integers.
- To overcome this, we can encode the integer in binary form. For example, if we assume the sequence length is $n = 8$, we can use the following sequence of $d = 3$-dimensional bit vectors for each location: 000, 001, 010, 011, 100, 101, 110, 111.
- We see that the right most index toggles the fastest (has highest frequency), whereas the left most index (most significant bit) toggles the slowest.
- We can represent this as a position matrix $\mathbf{P} \in \mathbb{R}^{n \times d}$.
- We can think of the above representation as using a set of basis functions (corresponding to powers of 2), where the coefficients are 0 or 1.
- We can obtain a more compact code by using a different set of basis functions, and real-valued weights.

Positional encoding (2/3)

- For example, it is common to use a sinusoidal basis, as follows:

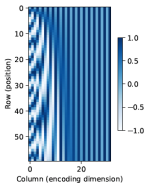
$$p_{i,2j} = \sin\left(\frac{i}{C^{2j/d}}\right), \quad p_{i,2j+1} = \cos\left(\frac{i}{C^{2j/d}}\right)$$

  where $C = 10,000$ corresponds to some maximum sequence length.

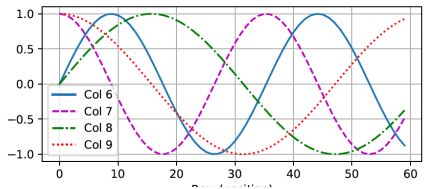
- For example, if $d = 4$, the i'th row is

$$\mathbf{p}_i = \left[\sin\left(\frac{i}{C^{0/4}}\right), \cos\left(\frac{i}{C^{0/4}}\right), \sin\left(\frac{i}{C^{2/4}}\right), \cos\left(\frac{i}{C^{2/4}}\right)\right]$$

- The figure (a) below shows the corresponding position matrix for $n = 60$ and $d = 32$. In this case, the left-most columns toggle fastest. We see that each row has a real-valued "fingerprint" representing its location in the sequence. Positional Encoding Tutorial.
- The figure (b) shows some of the basis functions (column vectors) for dimensions 6 to 9.



(a)



(b)

Positional encoding (3/3)

- The advantage of this representation is two-fold. First, it can be computed for arbitrary length inputs (up to $T \leq C$), unlike a learned mapping from integers to vectors.
- Second, the representation of one location is linearly predictable from any other, given knowledge of their relative distance.
- In particular, we have $\mathbf{p}_{t+\phi} = f(\mathbf{p}_t)$, where f is a linear transformation.
- To see this, note that

$$\begin{aligned}\begin{pmatrix}\sin(\omega_k(t+\phi)) \\ \cos(\omega_k(t+\phi))\end{pmatrix} &= \begin{pmatrix}\sin(\omega_k t)\cos(\omega_k\phi) + \cos(\omega_k t)\sin(\omega_k\phi) \\ \cos(\omega_k t)\cos(\omega_k\phi) - \sin(\omega_k t)\sin(\omega_k\phi)\end{pmatrix} \\ &= \begin{pmatrix}\cos(\omega_k\phi) & \sin(\omega_k\phi) \\ -\sin(\omega_k\phi) & \cos(\omega_k\phi)\end{pmatrix}\begin{pmatrix}\sin(\omega_k t) \\ \cos(\omega_k t)\end{pmatrix}\end{aligned}$$

- Once we have computed the positional embeddings $\mathbf{P}$, we need to combine them with the original word embeddings $\mathbf{X}$ using the following:

$$\text{POS}(\text{Embed}(\mathbf{X})) = \mathbf{X} + \mathbf{P}$$

Putting it all together (encoder)

- A transformer is a seq2seq model that uses self-attention for the encoder and decoder rather than an RNN. The encoder uses a series of encoder blocks, each of which uses multi-headed attention, residual connections, feed-forward layers, and layer normalization:

```
def EncoderBlock(X):
    Z = LayerNorm(MultiHeadAttn(Q=X, K=X, V=X) + X)
    E = LayerNorm(FeedForward(Z) + Z)
    return E
```

- The overall encoder is defined by applying positional encoding to the embedding of the input sequence, following by N (depth) copies of the encoder block:

```
def Encoder(X, N):
    E = POS(Embed(X))
    for n in range(N):
        E = EncoderBlock(E)
    return E
```

Putting it all together (decoder)

- The decoder is given access to the encoder via another multi-head attention block. But it is also given access to previously generated outputs: these are shifted, and then combined with a positional embedding, and then fed into a masked (causal) multi-head attention model. Finally the output distribution over tokens at each location is computed in parallel.

```
def DecoderBlock(Y, E):
    Z1 = LayerNorm(MultiHeadAttn(Q=Y, K=Y, V=Y) + Y)
    Z2 = LayerNorm(MultiHeadAttn(Q=Z1, K=E, V=E) + Z1)
    D = LayerNorm(FeedForward(Z2) + Z2)
    return D
```

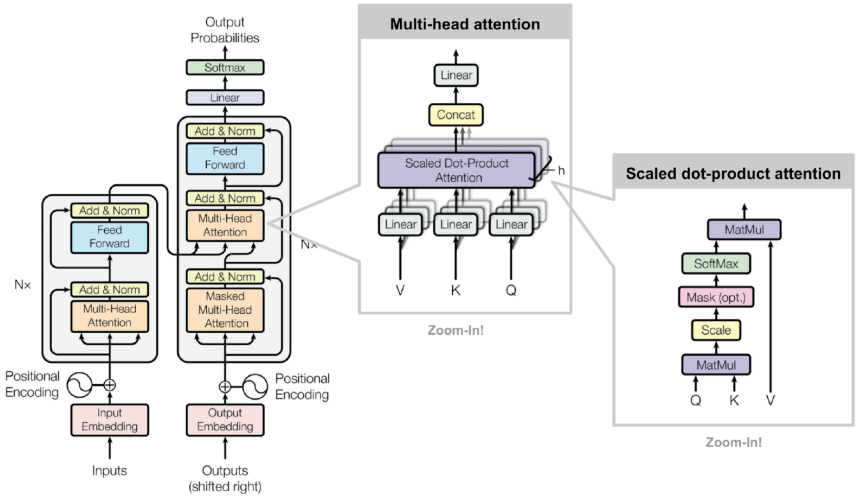
- The overall decoder is then defined by N copies of the decoder block:

```
def Decoder(Y, E, N):
    D = POS(Embed(Y))
    for n in range(N):
        D = DecoderBlock(D, E)
    return D
```

- During training time, all the inputs Y to the decoder are known in advance. At test time, we need to decode sequentially, and use masked attention, where we feed the generated output into the embedding layer, and add it to the set of keys/values that can be attended to.

Putting it all together (full architecture)

- Transformer Tutorial.



Efficient transformers

- Regular transformers take $\mathcal{O}(N^2)$ time and space complexity, for a sequence of length N, which makes them impractical to apply to long sequences. In the past few years, researchers have proposed several more efficient variants of transformers to bypass this difficulty.
- The simplest modification of the attention mechanism is to constrain it to sparse/localized window, restricting each token to attend only selected other tokens e.g. Reformer[15].
- Another approach approximates attention as

$$A_{ij} = \phi(\boldsymbol{q}_i)^\top \phi(\boldsymbol{k}_j)$$

  where $\phi(\boldsymbol{x}) \in \mathbb{R}^M$ is some finite-dimensional vector with $M < D$. One can leverage this structure to compute **AV** in $O(N)$ time.
- In Linformer[16], they transform the keys and values via random Gaussian projections via Johnson-Lindenstrauss theory.
- In Performer[17], they show that the attention matrix can be computed using a (positive definite) kernel function which also simplifies computations to in $O(N)$ time.

[15] N. Kitaev et al. "Reformer: The Efficient Transformer". In: ICLR 2020.

[16] S. Wang et al. "Linformer: Self-Attention with Linear Complexity". In: CoRR (2020).

[17] K. Choromanski et al. "Rethinking Attention with Performers". In: CoRR (2020).

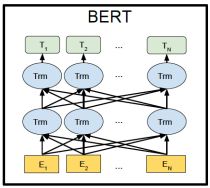
4) Generative modelling for text

Language models

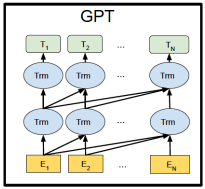
- We have discussed how RNNs and autoregressive (decoder-only) transformers can be used as language models, which are generative sequence models of the form

$$p(x_1, \ldots, x_T) = \prod_{t=1}^{T} p(x_t | x_{1:t-1})$$

  where each x_t is a discrete token, such as a word.
- This approach is usally carried out in two phases: we first pre-train the language model in an unsupervised way, on a large corpus of text, and then we fine-tune the model in a supervised way on a small labelled task-specific dataset.
- If our primary goal is to compute useful text representations, as opposed to generating text, we can replace the generative sequence model with non-causal models that can compute a representation of a sentence, but cannot generate it. These models have the advantage that now the hidden state $\boldsymbol{h}_t$ can depend on the past, $\boldsymbol{y}_{1:t-1}$, present $\boldsymbol{y}_t$, and future, $\boldsymbol{y}_{t+1:T}$.
- If we want to generate text, we must use a "causal" or generative language model.



(a)



(b)

BERT

- Bidirectional Encoder Representations from Transformers (BERT)[18] is a non-causal model, that can be used to create representations of text (but not to generate text).
- It uses a transformer model to map a modified version of a sequence back to the unmodified form. The modified input at location t omits the t'th token, and the task is to predict the missing word given the remaining words.
- More precisely, the model is trained to minimize the negative log pseudo-likelihood:

$$\mathcal{L} = \mathbb{E}_{\boldsymbol{x}\sim\mathcal{D}}\mathbb{E}_m \sum_{i\in m} -\log p(x_i \mid \boldsymbol{x}_{-m})$$

  where $\boldsymbol{m}$ is a random binary mask.
- For example, we might use a training sentence of the form

  Let's make [MASK] chicken! [SEP] It [MASK] great with orange sauce.

  where [SEP] is a separator token inserted between two sentences. The desired target labels for the masked words are "some" and "tastes".
- The conditional probability is given by applying a softmax

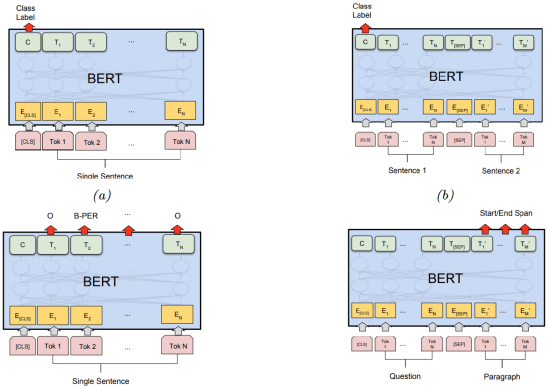
$$p(x_i \mid \hat{\boldsymbol{x}}) = \frac{\exp(\boldsymbol{h}(\hat{\boldsymbol{x}})_i^\top \boldsymbol{e}(x_i))}{\sum_{x'} \exp(\boldsymbol{h}(\hat{\boldsymbol{x}})_i^\top \boldsymbol{e}(x'))}$$

  where $\hat{\boldsymbol{x}} = \boldsymbol{x}_{-m}$ is the masked input sentence, and $\boldsymbol{e}(x)$ is the embedding for token x.
- We then compute the loss at masked locations; therefore called a masked language model.

[18] J. Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: NAACL. 2019.

Fine-tuning BERT

- After pre-training BERT in an unsupervised way, we can use it for various downstream tasks by performing supervised fine-tuning.
- The next figure illustrates how we can modify a BERT model to perform different tasks, by simply adding one or more new output heads to the final hidden layer.



- (a) Single sentence classification (e.g., sentiment analysis); (b) Sentence-pair classification; (c) Single sentence tagging; (d) Question answering. BERT Tutorial.

Introduction to LLMs (1/2)

- Here we briefly discuss generative or causal language models based on (decoder-only) transformers. These are generally called large language models (LLMs).
- Generative Pre-training Transformer (GPT)[19] is a causal (generative) model, that uses a masked transformer as the decoder.
- In the original GPT paper, they jointly optimize on a large unlabelled dataset, and a small labelled dataset. In the classification setting, the loss is given by

$$\mathcal{L} = \mathcal{L}_{\text{cls}} + \lambda \mathcal{L}_{\text{LM}},$$

where $\mathcal{L}_{\text{cls}} = -\sum_{(x,y)\in\mathcal{D}_L} \log p(y \mid x)$ is the classification loss on the labeled data, and

$$\mathcal{L}_{\text{LM}} = -\sum_{x\in\mathcal{D}_U} \sum_{t} \log p(x_t \mid x_{1:t-1})$$

is the language modeling loss on the unlabelled data.

- GPT-2[20] is a larger version of GPT, trained on a large web corpus called WebText; it eliminates any task-specific training, and instead it is just trained as a language model.
- GPT-3[21] and GPT-4[22] models are larger versions of GPT-2, but based on same principles.

[19] A. Radford et al. Improving Language Understanding by Generative Pre-Training. Tech. rep. OpenAI, 2018.

[20] A. Radford et al. Language Models are Unsupervised Multitask Learners. Tech. rep. OpenAI, 2019.

[21] T. B. Brown et al. "Language Models are Few-Shot Learners". In: (2020). arXiv: 2005.14165

[22] OpenAI. GPT4. Tech. rep. 2023.

Introduction to LLMs (2/2)

- More recently, OpenAI released ChatGPT[23], which is an improved version of GPT-3 which has been trained to have interactive dialogs by using a technique called reinforcement learning from human feedback (RLHF).
- This uses reinforcement learning techniques to fine tune the model so that it generates responses that are more "aligned" with human intent, as estimated by a ranking model, which is pre-trained on supervised data.

[23] OpenAI. ChatGPT: Optimizing Language Models for Dialogue. Blog.

Thank you for your attention!