Neural Networks

Personal notes from UIC CS 559, Fall 2025

Himanshu Dongre

Based on lectures by Erdem Koyuncu

These are personal notes written while I was enrolled in CS 559: Neural Networks at the University of Illinois Chicago in Fall 2025. The course was taught by Prof. Erdem Koyuncu. These are not official course materials and are not endorsed by the instructor or the University of Illinois Chicago. The organization, annotations, and any errors are my own.

Short quotations attributed to Prof. Erdem Koyuncu were recorded during Fall 2025 lectures. Instructor-derived explanations have been reorganized and annotated as part of my personal course compendium; they should not be read as verbatim course materials unless set off as a quotation. No blanket publication license is asserted over third-party material.

1 Prologue

Neural means related to neurons. A biological neuron is a nerve cell in the human body. It is a highly specialized cell that can transmit and receive information through electrical and chemical signals. A *network* is a set of nodes connected to each other. Hence, a *neural network* contains neurons with connections between them. Humans have *biological neural networks (BNNs)* in their bodies (CNS, PNS, etc.). We mathematically model *artificial neural networks (ANNs)* taking inspiration from BNNs. ANNs came up as a way of doing machine learning for tasks like classification, object recognition, prediction, etc. These tasks are trivial for BNNs but it's relatively very difficult to design good learning algorithms for the same tasks for ANNs. Compare BNNs (human brain) and ANNs (computer) across some parameters:

- **Speed:** The unit of a BNN is a nerve cell. An ANN's mathematical unit is an artificial neuron, implemented through many operations on processors rather than by a single transistor. Individual transistor switching can occur on nanosecond scales, while biological neural signaling and synaptic transmission occur on millisecond scales; the comparison is only a rough motivation, not a like-for-like benchmark.
- **Parallelism:** A computer is limited in how much the operations can be parallelized, although modern CPUs, GPUs, and specialized accelerators execute many operations concurrently. A BNN is highly parallel, with many signals propagating simultaneously. GPUs boost parallelism in computers.
- **Learning:** In ordinary training, an ANN's chosen computational graph is usually fixed while its parameters change, although architectures can also be changed or learned. On the other hand, BNNs and strengths of connections in them evolve over time.

We model ANNs as having some inputs $x_1, x_2, \ldots, x_n$ and some outputs $y_1, y_2, \ldots, y_m$ (a simplistic model of how humans operate). Some properties of ANNs:

- **Nonlinearity:** A *linear system* is one which can express its outputs as linear combinations of the inputs. If $\vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$ and $\vec{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}$, then a linear relation is expressed as $\vec{y} = A\vec{x} + \underbrace{\vec{b}}_{\text{bias}}$. ANNs are nonlinear systems.
- **Learning:** All connections between neurons have different strengths mathematically modeled by numerical *weights*. The process of learning involves changing the weights of a neural network so that it does the task that we want it to do. Each weight is a *parameter* of the neural network. The algorithm that we use to assign values to the parameters is the *learning algorithm*. Learning algorithms can be:
  1. **Supervised:** with a teacher. Weights are adjusted based on feedback.
  2. **Unsupervised:** without a teacher.

- **Adaptivity:** We can adapt a neural network to changes in the environment, changes in the task, changes in the data, etc. by changing the weights accordingly.
- **Evidential response:** aka *confidence level*. Instead of a hard decision, a neural network can be made to output a probability score.
- **Fault tolerance:** Some neural networks degrade gracefully under limited neuron or connection loss, depending on the architecture, redundancy, training, and which units are removed; removal of many or important units can destroy functionality.
- **Neurobiological analogy:** Both neurobiologists and computer scientists gain from studying ANNs.

The way we approach the study of neural networks is by (1) finding a mathematical model for one neuron, (2) building networks of neurons, and (3) designing learning algorithms so that the network performs a desired task.

2 The Neuron

The mathematical model for one artificial neuron is inspired from a biological neuron. The transmission medium between the axon of a neuron and a dendrite of another neuron is a *synapse*. A synaptic connection can excite or inhibit a neuron. We model the strengths of synaptic connections with numerical weights. There are $\sim 10^{14}$ synapses in the human brain, which is 100 trillion parameters (compare with the strongest LLMs which have 100s of billions of parameters). We think of $x_1, x_2, \ldots, x_n$ as inputs coming in to the soma (cell body) of a neuron, where each input x_i is multiplied by a weight w_i at the synaptic connection. The full signal is the weighted sum of the inputs. The transmission of the signal through the axon is modeled using a nonlinearity. The signal ultimately exits through the axon terminals which are nothing but inputs to other neurons. Hence, the entity of interest in the neuron is the signal and we model that as the output of the neuron. We include the bias b in the model by hardwiring the input $x_0 = 1$. Let $\vec{w} = \begin{bmatrix} b \\ w_1 \\ \vdots \\ w_n \end{bmatrix}$, φ be the nonlinearity, $\vec{x}$ be the vector of inputs as before (now including x_0), and y be the signal or the output of the neuron. Then a neuron is mathematically modeled as:

$$y = \varphi(\vec{w}^{\mathsf{T}}\vec{x})\,.$$

The signal before passing through the nonlinearity is called the **induced local field (ILF)** or simply the local field. Hence the local field of a neuron is:

$$a = \vec{w}^{\mathsf{T}}\vec{x}\,.$$

The output of the neuron is the activation of its local field:

$$y = \varphi(a)\,.$$

The rough count above should not be interpreted as 100 trillion ANN parameters: biological synapses are not one-for-one equivalents of learned artificial-network parameters, and comparisons with language-model parameter counts are only suggestive.

2.1 Activation Functions

There are many choices for the activation function in our mathematical model:

1. **Heaviside step function:** $\varphi(a) = u(a) = \begin{cases} 1 & \text{if } a \geq 0 \\ 0 & \text{otherwise} \end{cases}$.
2. **Sigmoid function:** $\varphi(a) = \sigma_\alpha(a) = \frac{1}{1+e^{-\alpha a}}$, where α is a parameter such that $\alpha > 0$.
   - $\alpha = 1$ gives a continuous version of the step function.
   - $\frac{d\sigma_\alpha}{da} = -(1+e^{-\alpha a})^{-2}(-\alpha e^{-\alpha a}) = \alpha \cdot \frac{1}{1+e^{-\alpha a}} \cdot \frac{e^{-\alpha a}}{1+e^{-\alpha a}} = \alpha\sigma_\alpha(1-\sigma_\alpha)$. As α increases, σ_α becomes more compressed and steeper in the middle. As α decreases, it becomes flatter. $\alpha \to \infty$ gives convergence to the Heaviside function except at 0.
3. **Signum (sign) function:** $\varphi(a) = \begin{cases} +1 & \text{if } a \geq 0 \\ -1 & \text{if } a < 0 \end{cases}$.
4. **Hyperbolic tangent:** $\varphi(a) = \tanh_\alpha(a) = \frac{e^{\alpha a} - e^{-\alpha a}}{e^{\alpha a} + e^{-\alpha a}}$, where α is a parameter such that $\alpha > 0$.
   - $\alpha = 1$ gives a continuous version of the signum function.
   - As α increases, $\tanh_\alpha$ becomes more compressed and steeper in the middle. As α decreases, it becomes flatter. $\alpha \to \infty$ gives convergence to the signum function except at 0.
5. **Rectified linear unit (ReLU):** $\varphi(a) = \max(0, a)$.
   - The linear unit is $\varphi(a) = a$ but we need to *rectify* it to make it nonlinear.
   - ReLU is not differentiable at 0.
6. **Gaussian error linear unit (GELU):** $\varphi(a) = a\Phi(a)$, where $\Phi(a)$ is the CDF of the standard normal distribution.
   - GELU weights inputs by their probability of being positive, under a standard normal. This is unlike ReLU which hard-thresholds at 0. GELU allows small negative values to pass through but dampens them.
   - GELU is smooth and differentiable everywhere.

Other variants of ReLU like ELU and Swish are also prevalent.

3 Neural Network Architectures

This section provides examples of how some common neural network architectures are assembled together from individual neurons.

3.1 Single-Layer Feed-Forward Neural Network

Until now, we have mathematically modeled a single neuron. Nobody prevents us from adding more neurons that take the same inputs. Suppose instead of one, we have m neurons such that all the m neurons take all n inputs and have the same activation φ. All the m neurons will have:

- their own weight vectors $\vec{w}_1, \ldots, \vec{w}_m$ (biases excluded for now),
- their own biases $b_1, \ldots, b_m$,
- their own ILFs $a_1, \ldots, a_m$ such that $a_i = \vec{w}_i^\mathsf{T} \vec{x}$, resulting in their own signals,
- their own outputs $y_1, \ldots, y_m$ such that $y_i = \varphi(a_i)$.

In this architecture we refer to the inputs as forming the *input layer* and the neurons as forming the *output layer* because they ultimately give us the outputs. Thus, we have our first neural network architecture: the *single-layer feed-forward neural network* that takes in inputs $x_1, \ldots, x_n$ and gives outputs $y_1, \ldots, y_m$.

- It is single-layer because we do not count the input units as a layer of neurons; we count the one output layer of neurons.
- It is a feed-forward network because information flows in one direction—from the inputs to the outputs. We provide the inputs, the neural network does some operations and returns the outputs. It is possible to have *feedback* signals from the output layer to the input layer. That would make the network a *feedback network* and not feed-forward.

We define the following mathematical objects:

$$\vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \; W = \begin{bmatrix} \vec{w}_1^\mathsf{T} \\ \vec{w}_2^\mathsf{T} \\ \vdots \\ \vec{w}_m^\mathsf{T} \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m1} & w_{m2} & \cdots & w_{mn} \end{bmatrix}, \; \vec{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}, \; \vec{a} = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{bmatrix}, \; \vec{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}.$$

Then we model the input-output relationship of the single-layer feed-forward neural network as follows:

$$\begin{aligned} \vec{a} &= W\vec{x} + \vec{b}\,, \\ \vec{y} &= \varphi(\vec{a})\,, \end{aligned}$$

where the activation φ now acts element-wise on $\vec{a}$. Thus our complete model for the single-layer feed-forward neural network is:

$$\boxed{\vec{y} = \varphi(W\vec{x} + \vec{b})}\,.$$

3.2 Multi-layer Feed-Forward Neural Networks

The outputs from a single-layer feed-forward neural network can be passed as inputs to a second layer of neurons with the same activation. Let's characterize the second layer with p neurons, its weight matrix U, its bias vector $\vec{c}$, and its output vector $\vec{z}$. Then we can write the input-output relationship for the second layer as:

$$\vec{z} = \varphi(U\vec{y} + \vec{c}) \,.$$

Keep in mind the dimensions:

$\vec{x} : n \times 1,\ W : m \times n,\ \vec{b} : m \times 1,\ \vec{y} : m \times 1,\ U : p \times m,\ \vec{c} : p \times 1,\ \vec{z} : p \times 1 \,.$

Thus the complete mathematical model for the two-layer neural network is:

$$\vec{z} = \varphi(U\varphi(W\vec{x} + \vec{b}) + \vec{c}) \,.$$

This neural network is still a feed-forward network because we haven't added any feedback and the flow of information is maintained in one direction. In this architecture, the inputs form the input layer as before, and the *last* layer—the U-layer—is the output layer because it spits out the outputs. We have an additional layer—the W-layer—which is called the *hidden layer* because it is abstracted as a black box between the inputs and the outputs. Under the counting convention used here, input units are not counted, while the hidden and output layers are. A **shallow neural network** refers to a neural network with one hidden layer. We can add more cascading layers to the neural network and all but the last will be considered hidden layers. A neural network with more than one hidden layer is a **deep neural network** under this convention, but there is no universal numerical cutoff such as ten hidden layers.

3.3 Single-Layer Feedback Neural Network

The simplest example of a feedback neural network is to take a single-layer feed-forward neural network with $m = n$ and add connections from the output layer to the input layer. We need a notion of time to describe the flow of information in such a network. At time $t = 0$ we pass in the input $\vec{x}$ and obtain $\vec{y}_1 = \varphi(W\vec{x} + \vec{b})$; at the next time step, $\vec{y}_2 = \varphi(W\vec{y}_1 + \vec{b})$. This recursion does not in general converge, and an arbitrary feedback network need not have an energy function. For the classical binary Hopfield model, an energy-decrease argument applies when weights are symmetric, diagonal weights are zero, and neurons are updated asynchronously with a deterministic threshold rule and a consistent tie convention. Under those assumptions, the finite-state dynamics reach a fixed point. Stored patterns can be fixed points/local energy minima, so a corrupted pattern such as `catt` may converge to a nearby stored pattern such as `cat`, but successful error correction is not guaranteed. This is the associative-memory motivation. A fixed point is a state for which further updates leave the state unchanged.

4 Perceptron

Perceptron is the legacy name for a single neuron with a step activation, $\varphi(a) = u(a) = \begin{cases} 1 & \text{if } a \geq 0 \\ 0 & \text{otherwise} \end{cases}$. The input-output relation for the perceptron is:

$$y = \varphi(\vec{w}^{\mathsf{T}}\vec{x} + b)\,.$$

The perceptron can simulate logic gates:

- Logical NOT: $y = u(-1.2x + 1)$, where $x \in \{0, 1\}$. This perceptron gives the following relation:

x	y
0	1
1	0

  which is precisely the logical NOT relation.

- Logical AND: $y = u\left(\begin{bmatrix}0.6\\0.6\end{bmatrix}^{\mathsf{T}}\begin{bmatrix}x_1\\x_2\end{bmatrix} - 1\right)$, where $\vec{x} \in \{0, 1\}^2$. This perceptron gives the following relation:

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

  which is precisely the logical AND relation. A general solution for $n \geq 2$ is $w_1 = w_2 = \ldots = w_n = \frac{1}{n} + \varepsilon$ for some small $\varepsilon \to 0^+$, and $b = -1$.

- Logical OR: $y = u\left(\begin{bmatrix}1\\1\end{bmatrix}^{\mathsf{T}}\begin{bmatrix}x_1\\x_2\end{bmatrix} - 1\right)$, where $\vec{x} \in \{0, 1\}^2$. This perceptron gives the following relation:

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

  which is precisely the logical OR relation. A general solution for $n \geq 2$ is $w_1 = w_2 = \ldots = w_n = 1$, and $b = -1$.

As a result, we can do any sums of products using perceptrons. We can implement any logic operation/gate using perceptrons only. So we can implement a computer using perceptrons only. It can further be shown that any logic gate can be implemented with at most two layers. Therefore a full digital computer can be implemented with at most two layers. A result of this kind is called a **universality result**. However, famously, the logical XOR gate cannot be implemented using a single perceptron. A natural question arises that which operations can be implemented using a perceptron and which cannot. To answer this, let's look at the general case:

$$y = u(\vec{w}^\mathsf{T}\vec{x} + \vec{b}) = \begin{cases} 1, & \vec{w}^\mathsf{T}\vec{x} + \vec{b} \geq 0 \\ 0, & \vec{w}^\mathsf{T}\vec{x} + \vec{b} < 0 \end{cases}.$$

We can plot the regions $\vec{w}^\mathsf{T}\vec{x} + \vec{b} \geq 0$ and $\vec{w}^\mathsf{T}\vec{x} + \vec{b} < 0$ to ascertain where the perceptron gives outputs 1 and 0 respectively. The regions are disjoint and cover the entire space. The boundary that separates the two regions is the curve $\vec{w}^\mathsf{T}\vec{x} + \vec{b} = 0$, which is a hyperplane. Thus, a perceptron is a **linear classifier**. The orientation of the decision boundary can be set by tweaking the weights and the bias.

4.1 Perceptron Training Algorithm

The PTA is a *learning algorithm* for the perceptron. A learning algorithm is a systematic approach towards computing the weights that yield the desired outputs for a given dataset.

Problem: Given dataset $\mathcal{D} = \{(\vec{x}_i, y_i)\}_{i=1}^n \subset \mathbb{R}^d \times \{0, 1\}$ that specifies the desired outputs y_i's for the inputs $\vec{x}_i$'s. Suppose a perceptron exists that can separate the two classes; i.e. there exists a $\vec{w}$ such that $y_i = u(\vec{w}^\mathsf{T}\vec{x}_i)$ (i.e. the classes are *linearly separable*). Find $\vec{w}$.

Perceptron training algorithm (PTA):

Set a value for the learning parameter $\eta > 0$.

1. Initialize $\vec{w}$ randomly.
2. epoch $\leftarrow 0$ `// an epoch is a full pass over the training dataset`
3. **while** $\vec{w}$ cannot classify all inputs correctly:
    1. epoch $\leftarrow$ epoch $+ 1$
    2. **for** $i = 1$ to n:
        1. $y \leftarrow u(\vec{w}^\mathsf{T}\vec{x}_i)$
        2. **if** $y_i = y$:
            1. pass `// do nothing`

           else if $y_i = 1$ and $y = 0$
            1. $\vec{w} \leftarrow \vec{w} + \eta\vec{x}_i$

           else `// desired output is 0 but y = 1`
            1. $\vec{w} \leftarrow \vec{w} - \eta\vec{x}_i$

           end

    end

end
4. **return** $\vec{w}$

Theorem. For a finite training set that is strictly linearly separable after incorporating a bias through an augmented coordinate, the PTA makes only finitely many mistakes for any fixed $\eta > 0$ (with the corresponding scaling absorbed into the separating weights).

The main PTA equation comes from the weights update which can be written in one shot as:

$$\vec{w} \leftarrow \vec{w} + (y_i - u(\vec{w}^\mathsf{T}\vec{x}_i))\eta\vec{x}_i \,.$$

Using this, the PTA can be written in condensed form:

1. Initialize $\vec{w}$ randomly.
2. **while** $\vec{w}$ cannot classify all inputs correctly:
    1. **for** $i = 1$ to n:
        1. $\vec{w} \leftarrow \vec{w} + (y_i - u(\vec{w}^\mathsf{T}\vec{x}_i))\eta\vec{x}_i$

    end

    end
3. **return** $\vec{w}$

5 Supervised Learning

The general structure for supervised learning is:

1. Initialize weights $\vec{w}$.
2. **for** epochs $= 1$ until convergence
    1. **for** $i = 1$ to $|\mathcal{D}|$
        1. Update $\vec{w}$.

In terms of how we update the weights, there are two approaches:

1. **Step-wise/Online learning:**

    Suppose at the beginning of an epoch, weights are set as $\vec{w}$.

    1. **for** $i = 1$ to $|\mathcal{D}|$
        1. Show $\vec{x}_i$, calculate output
        2. Compute weights update Δ_i
        3. $\vec{w} \leftarrow \vec{w} + \Delta_i$

    This loop is executed for every epoch. An example of online learning is the PTA.

2. **Batch/Offline learning:**

    In online learning, we compute the weights update in each step and also apply the weights. In offline learning, we compute the weights update in each step but don't update the weights in each step. We update the weights cumulatively. Typically, batch learning involves a *batch size*. Suppose the batch size is b and at the beginning of an epoch, weights are set as $\vec{w}$.

    1. **for** batch in batches($\mathcal{D}$)
        1. **for** $i = 1$ to b

1. Show $\vec{x}_i$, calculate output
2. Compute and store weights update Δ_i
   `// do not update weights`

end

2. $\vec{w} \leftarrow \vec{w} + \frac{1}{b}\sum_{i=1}^{b} \Delta_i$

We use the updated weights for each new batch instead of each new step.

5.1 General Formulation

We have a dataset (with desired outputs), $\mathcal{D} = \{(\vec{x}_i, \vec{y}_i)\}_{i=1}^{n} \subset \mathbb{R}^d \times \mathbb{R}^m$. We have a neural network that models a *neural network transfer function*, $\vec{x} \mapsto f(\vec{x}, \mathcal{W})$, where $\mathcal{W}$ encapsulates *all* the weights/parameters in the neural network.

The goal of supervised learning is to make the neural network outputs as close to the desired outputs as possible over $\mathcal{D}$. We want $f(\vec{x}_i, \mathcal{W})$ as close to $\vec{y}_i$ as possible for every $1 \leq i \leq n$. To check how far we are from the goal, we need some way to measure closeness of outputs. The simplest way is to take the mean squared Euclidean distance:

$$\ell(\mathcal{W}) = \frac{1}{n}\sum_{i=1}^{n} \|\vec{y}_i - f(\vec{x}_i, \mathcal{W})\|_2^2 \,.$$

This is a **loss function** that represents how much the NN outputs deviate from the desired outputs. Based on this, we can formally interpret the goal of supervised learning as minimizing the loss function.

5.2 Linear Regression

"This is machine learning 1, not even 101, 1."

— Prof. Erdem Koyuncu, Fall 2025 lecture

The simplest case is when we obtain $\arg\min_{\mathcal{W}} \ell$ in closed form. This happens when the transfer function is linear. A linear transfer function is given by a single-layer neural network with a linear/no activation characterized by a weight matrix $W \in \mathbb{R}^{m \times d}$. This NN gives an m-dimensional output or equivalently m outputs. The transfer function is:

$$f(\vec{x}, W) = W\vec{x} \,.$$

Substituting this in the mean squared Euclidean norm loss function, we get

$$\ell(W) = \frac{1}{n}\sum_{i=1}^{n} \|\vec{y}_i - W\vec{x}_i\|_2^2 \,.$$

Our goal is to find a solution

$$W^* = \arg\min_{W} \ \ell \,.$$

This problem is called **linear regression**. Define $X_{d\times n} = \begin{bmatrix}\vec{x}_1 & \vec{x}_2 & \cdots & \vec{x}_n\end{bmatrix}$ and $Y_{m\times n} = \begin{bmatrix}\vec{y}_1 & \vec{y}_2 & \cdots & \vec{y}_n\end{bmatrix}$. Then,

$$\ell(W) = \|Y - WX\|_F^2$$
$$\Rightarrow W^* = \arg\min_W \|Y - WX\|_F^2 .$$

We safely drop the constant factor $\frac{1}{n}$ from the optimization. A least-squares minimizer need not make the residual zero. The normal equation is

$$W^* X X^\mathsf{T} = Y X^\mathsf{T} .$$

Before giving the final solution, we define a **Moore-Penrose pseudoinverse (MPP)**. Let $A_{m\times n}$ be an arbitrary matrix. There exists a matrix $A^+_{n\times m}$ called the *Moore-Penrose pseudoinverse* of A such that:

1. $AA^+A = A$,
2. $A^+AA^+ = A^+$,
3. AA^+ and A^+A are symmetric.

If A has linearly independent rows, then $A^+ = A^\mathsf{T}(AA^\mathsf{T})^{-1}$. If A has linearly independent columns, then $A^+ = (A^\mathsf{T}A)^{-1}A^\mathsf{T}$.

Back to linear regression. For arbitrary X, the minimum-Frobenius-norm least-squares solution is $\boxed{W^* = YX^+}$. More generally, every least-squares solution has the form $W = YX^+ + Z(I - XX^+)$ for an arbitrary compatible matrix Z. If X has full row rank, this reduces to $W^* = YX^\mathsf{T}(XX^\mathsf{T})^{-1}$ and the fit is exact only when Y lies in the row space of X. If X has full column rank, $X^+ = (X^\mathsf{T}X)^{-1}X^\mathsf{T}$; the least-squares residual may still be nonzero. Rank independence is a condition to check, not an automatic machine-learning assumption.

6 Nonlinear Optimization

In linear regression, we obtained a closed-form solution for the loss function optimization when the transfer function was linear and the neural network had a linear/no activation. Typically, the neural network has a nonlinear activation and the transfer function is nonlinear: $f(\vec{x}, W) = \varphi(W\vec{x})$. Hence, the loss function becomes:

$$\ell(W) = \frac{1}{n}\sum_{i=1}^{n} \|\vec{y}_i - \varphi(W\vec{x}_i)\|_2^2 .$$

Such nonlinear optimization problems generally do not have convenient closed-form solutions, although special cases can. We typically resort to iterative optimization techniques.

6.1 Gradient Descent

Suppose we have a multivariate function $f(\vec{x})$ and we want to minimize it. We can compute the gradient $\vec{g} = \vec{\nabla} f$. $\vec{g}_{\vec{x}_0}$ gives us the direction at $\vec{x} = \vec{x_0}$ along which f increases the fastest (steepest ascent). Hence, the direction along the negative gradient $(-\vec{g}_{\vec{x}})$ is the direction along which f decreases the fastest (steepest descent). The *gradient descent* algorithm makes use of this fact to minimize a function by repeatedly taking steps (of some size) along the direction of the negative (local) gradient until it is close to or at the minimum.

Gradient descent algorithm (GD):

Set a value for the step size $\eta > 0$ and the stopping cutoff $\varepsilon > 0$.

1. $\vec{x} \leftarrow \vec{x}_{\text{init}}$
2. **repeat**
    1. $\vec{g} \leftarrow \vec{\nabla}_{\vec{x}} f$
    2. $\vec{x} \leftarrow \vec{x} - \eta \vec{g}$ `// GD update`

   until $\|\vec{g}\| \leq \varepsilon$
3. **return** $\vec{x}$

Unlike the PTA under its separability assumptions, the convergence of GD depends on the objective and step-size rule. If the step size is too high, GD may diverge. Under suitable smoothness and step-size assumptions, GD may converge to a stationary point; that point can be a local minimum, a saddle, or, in special cases, a global minimum.

The rationale behind choosing the negative gradient as the direction to step along comes from the first order Taylor approximation of f at a point $\vec{x}$:

$$f\left(\vec{x} + \vec{\delta}\right) \approx f(\vec{x}) + \vec{\delta}^{\mathsf{T}} \vec{g}\,.$$

This is a good approximation for small enough $\|\vec{\delta}\|$'s. Out of all the possible directions ($\vec{\delta}$'s) to step along from $\vec{x}$, we choose the one along which the function value decreases the most:

$$\begin{aligned}
\vec{\delta}^* &= \underset{\|\vec{\delta}\| \leq \rho}{\arg\min}\, f\left(\vec{x} + \vec{\delta}\right) \\
&\approx \underset{\|\vec{\delta}\| \leq \rho}{\arg\min}\, \underbrace{f(\vec{x})}_{\text{constant}} + \vec{\delta}^{\mathsf{T}} \vec{g}_{\vec{x}} \\
&= \underset{\|\vec{\delta}\| \leq \rho}{\arg\min}\, \vec{\delta}^{\mathsf{T}} \vec{g}_{\vec{x}} \\
&= -\rho \frac{\vec{g}_{\vec{x}}}{\|\vec{g}_{\vec{x}}\|} \quad (\vec{g}_{\vec{x}} \neq 0)\,.
\end{aligned}$$

Thus the constrained steepest-descent direction is the negative gradient direction. Writing the step as $-\eta \vec{g}_{\vec{x}}$ chooses $\rho = \eta \|\vec{g}_{\vec{x}}\|$, small enough for the linear Taylor approximation to be

useful. Hence, in one step/iteration of GD, the following updates happen:

$$\begin{aligned}\vec{x} &\leftarrow \vec{x} + \vec{\delta} \\ &= \vec{x} - \eta \vec{g}_{\vec{x}}\,,\end{aligned}$$

$$\begin{aligned}f(\vec{x}) &\leftarrow f(\vec{x}) + \vec{\delta}^{\mathsf{T}} \vec{g}_{\vec{x}} \\ &= f(\vec{x}) + (-\eta \vec{g}_{\vec{x}})^{\mathsf{T}} \vec{g}_{\vec{x}} \\ &= f(\vec{x}) - \eta \left\| \vec{g}_{\vec{x}} \right\|^2 .\end{aligned}$$

Thus the decrease in the function value per step/iteration of GD is approximately $\eta \left\| \vec{g}_{\vec{x}} \right\|^2$.

6.2 Newton's Method

If we take a second order Taylor approximation of the objective then we get:

$$f\left(\vec{x} + \vec{\delta}\right) \approx f(\vec{x}) + \vec{\delta}^{\mathsf{T}} \vec{g}_{\vec{x}} + \frac{1}{2} \vec{\delta}^{\mathsf{T}} H_{\vec{x}} \vec{\delta},$$

where $h = \nabla^2 f$ is the *Hessian* matrix. This is a better approximation than the linear one we used in GD for small enough $\|\vec{\delta}\|$'s. Out of all the possible directions ($\vec{\delta}$'s) to step along from $\vec{x}$, we choose the one along which the function value decreases the most:

$$\begin{aligned}\vec{\delta}^* &= \arg\min_{\vec{\delta}} f\left(\vec{x} + \vec{\delta}\right) \\ &\approx \arg\min_{\vec{\delta}} f(\vec{x}) + \vec{\delta}^{\mathsf{T}} \vec{g}_{\vec{x}} + \frac{1}{2} \vec{\delta}^{\mathsf{T}} H_{\vec{x}} \vec{\delta}\,.\end{aligned}$$

Taking the derivative of the minimand w.r.t. $\vec{\delta}$ and setting it to zero, we get:

$$\begin{aligned}\vec{g}_{\vec{x}} + H_{\vec{x}} \vec{\delta}^* &\stackrel{\text{set}}{=} 0 \\ \Rightarrow \vec{\delta}^* &= -H_{\vec{x}}^{-1} \vec{g}_{\vec{x}}\,.\end{aligned}$$

If $H_{\vec{x}}$ is invertible, the ordinary Newton update uses $\eta = 1$: $\vec{x} \leftarrow \vec{x} - H_{\vec{x}}^{-1} \vec{g}_{\vec{x}}$. Interpreting the stationary point of the quadratic model as its minimizer additionally requires positive-definite curvature. Using $0 < \eta < 1$ is a damped Newton update, often combined with a line search or trust region. In practice one solves a linear system rather than explicitly forming H^{-1}.

6.3 Application

Suppose we have a supervised learning setup with a dataset $\mathcal{D} = \{(\vec{x}_i, y_i)\}_{i=1}^n$ (consider the outputs to be 1D for now). We know that the linear regression solution is given by

$\vec{w}^* = \vec{y}X^\mathsf{T}(XX^\mathsf{T})^{-1}$. The optimization approach we took to obtain this was using calculus-based methods. Let's optimize the loss using GD:

$$\ell(\vec{w}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \vec{w}^\mathsf{T}\vec{x}_i)^2$$
$$\Rightarrow \vec{\nabla}\ell = \frac{1}{n}\sum_{i=1}^{n} 2(y_i - \vec{w}^\mathsf{T}\vec{x}_i)(-\vec{x}_i)\,.$$

The GD update while optimizing the loss looks like:

$$\begin{aligned}\vec{w} &\leftarrow \vec{w} - \eta\vec{\nabla}\ell \\ &= \vec{w} + \frac{2\eta}{n}\sum_{i=1}^{n}(y_i - \vec{w}^\mathsf{T}\vec{x}_i)\vec{x}_i\,.\end{aligned}$$

This is the GD form of linear regression. Note that we have to sum over *all* data samples *per update*. Alternatively, we can view it as accumulating the weight updates over all the data samples and updating the weights cumulatively, which is akin to offline learning. We can come up with an online learning version of this by splitting one update into n different updates where the data samples are shown one at a time.

$$\vec{w} \leftarrow \vec{w} + \frac{2\eta}{n}(y_i - \vec{w}^\mathsf{T}\vec{x}_i)\vec{x}_i\,, \quad i = 1, \ldots, n\,.$$

This algorithm (in this form) to solve a linear system is called the **least mean squares (LMS)** algorithm. It is an approximation of the true GD algorithm.

Now suppose we apply a nonlinear activation to the local field in the neural network, so the output of the NN becomes $y = \varphi(\vec{w}^\mathsf{T}\vec{x})$. The loss becomes:

$$\ell(\vec{w}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \varphi(\vec{w}^\mathsf{T}\vec{x}_i))^2$$
$$\Rightarrow \vec{\nabla}\ell = \frac{1}{n}\sum_{i=1}^{n} 2(y_i - \varphi(\vec{w}^\mathsf{T}\vec{x}_i))(-\varphi'(\vec{w}^\mathsf{T}\vec{x}_i))\vec{x}_i\,.$$

The GD update will look like:

$$\begin{aligned}\vec{w} &\leftarrow \vec{w} - \eta\vec{\nabla}\ell \\ &= \vec{w} + \frac{2\eta}{n}\sum_{i=1}^{n}(y_i - \varphi(\vec{w}^\mathsf{T}\vec{x}_i))\varphi'(\vec{w}^\mathsf{T}\vec{x}_i)\vec{x}_i\,.\end{aligned}$$

Just like in the linear case, we can devise an online learning version of this by getting rid of the summation:

$$\begin{aligned}\vec{w} &\leftarrow \vec{w} + \frac{2\eta}{n}(y_i - \varphi(\vec{w}^\mathsf{T}\vec{x}_i))\varphi'(\vec{w}^\mathsf{T}\vec{x}_i)\vec{x}_i\,, \quad i = 1, \ldots, n\,, \\ &= \vec{w} + \frac{2\eta}{n}(y_i - \hat{y}_i)\varphi'(a_i)\vec{x}_i\,.\end{aligned}$$

In this form, the update is the **nonlinear delta rule**; Widrow-Hoff or LMS conventionally refers to the linear-unit case. For the parameterized sigmoid $y = \sigma_\alpha(a)$, $\varphi'(a) = \alpha y(1-y)$. The online update is therefore

$$\vec{w} \leftarrow \vec{w} + 2\eta(y_i - \hat{y}_i)\alpha\hat{y}_i(1-\hat{y}_i)\vec{x}_i\,.$$

Here the online learning rate absorbs any constant $1/n$ convention used in the batch mean loss. For $\tanh(\alpha a)$, the derivative is $\alpha(1-y^2)$.

7 Backpropagation

Until now we've seen how to optimize the objective for a single neuron with no/linear activation, and a single neuron with nonlinear activation. It's time to extend this to the general case. The **backpropagation algorithm (BPA)** efficiently calculates $\nabla_{\mathcal{W}}\ell$, after which a gradient-based optimizer can update the parameters. Suppose $\mathcal{D} = \{(\vec{x}_i, \vec{y}_i)\}_{i=1}^n$ and

$$\ell(\mathcal{W}) = \frac{1}{n}\sum_{i=1}^{n} \|\vec{y}_i - f(\vec{x}_i, \mathcal{W})\|_2^2\,.$$

For a k-layer network, use column vectors and keep biases separate: $\vec{a}_j = W_j\vec{z}_j + \vec{b}_j$ and $\vec{z}_{j+1} = \varphi(\vec{a}_j)$, with $\vec{z}_1 = \vec{x}$ and $\hat{\vec{y}} = \varphi(\vec{a}_k)$. Then we construct a feed-forward graph and a backward graph as follows:

1. Compute the output error: $\frac{\partial\ell}{\partial\hat{\vec{y}}_i} = -2\Big(\vec{y}_i - \hat{\vec{y}}_i\Big)$, where $\vec{y}_i$ is the target label for the data point $\vec{x}_i$, and $\hat{\vec{y}}_i$ is the prediction of the NN such that $\hat{\vec{y}}_i = f(\vec{x}_i, \mathcal{W})$.
2. For the per-example squared error, initialize the *error signal* at the output layer: $\vec{\delta}_k = 2\,\varphi'(\vec{a}_k) \odot \Big(\hat{\vec{y}}_i - \vec{y}_i\Big)$. (If the per-example loss includes a factor $1/2$, omit the factor 2.)
3. Propagate the error signal as a backward pass from the output layer to the input layer using the recursion

$$\vec{\delta}_j = \Big(W_{j+1}^{\mathsf{T}}\vec{\delta}_{j+1}\Big) \odot \varphi'(\vec{a}_j), \quad \text{for } j = k-1, \ldots, 1\,.$$

   The output of the feedback network is $\vec{\delta}_0 = W_1^{\mathsf{T}}\vec{\delta}_1$.

Key idea: the weight gradient is an outer product of the error signal and the layer input. Thus, for column-vector signals, $\boxed{\frac{\partial\ell_i}{\partial W_j} = \vec{\delta}_j\vec{z}_j^{\mathsf{T}}}$ and $\frac{\partial\ell_i}{\partial\vec{b}_j} = \vec{\delta}_j$ for $j = 1, \ldots, k$. For the mean loss, average these per-example gradients. For another differentiable loss, initialize the output signal with $\frac{\partial\ell_i}{\partial\hat{\vec{y}}_i} \odot \varphi'(\vec{a}_k)$.

7.1 Algorithm

Backpropagation is a standard efficient method for differentiating many neural networks, after which an optimizer may use those gradients. It is not the training method for every

neural network, and a hard-threshold perceptron is ordinarily trained by its own update rule. Consider the supervised learning setup and a differentiable neural network which we wish to train.

7.1.1 Online Learning

1. Initialize weights $\mathcal{W}$ (e.g.:- randomly)
2. **for** epochs 1 until convergence
    1. **for** $i = 1$ to n
        1. Take $\vec{x}_i$ with desired output $\vec{y}_i$
        2. $\hat{\vec{y}}_i \leftarrow$ forward pass
        3. $\nabla_{\mathcal{W}}\ell \leftarrow$ backward pass
        4. $\mathcal{W} \leftarrow \mathcal{W} - \eta\nabla_{\mathcal{W}}\ell$

The BPA is used in step 2.1.3 to do the backward pass with $\hat{\vec{y}}_i$ as the input.

7.1.2 Offline Learning

1. Initialize weights $\mathcal{W}$ (e.g.:- randomly)
2. **for** epochs 1 until convergence
    1. $\mathcal{G} \leftarrow 0$
    2. **for** $i = 1$ to n
        1. Take $\vec{x}_i$ with desired output $\vec{y}_i$
        2. $\hat{\vec{y}}_i \leftarrow$ forward pass
        3. $\nabla_{\mathcal{W}}\ell \leftarrow$ backward pass
        4. $\mathcal{G} \leftarrow \mathcal{G} + \nabla_{\mathcal{W}}\ell$
    3. $\mathcal{W} \leftarrow \mathcal{W} - \frac{\eta}{n}\mathcal{G}$

The BPA is used in step 2.2.3 to do the backward pass with $\hat{\vec{y}}_i$ as the input.

8 Training A Neural Network

> "With 4 parameters, I can fit an elephant; with 5, I can make it wiggle its trunk."
>
> — John Von Neumann

Many modern neural networks are trained with gradient-based optimizers and backpropagation, but neither method is universal.

8.1 Normalization

There is generally a data preprocessing step involved in training neural networks. We like our input data to be centered around zero and for it to be uncorrelated to achieve good training. We can transform the data so that it achieves these properties. This transformation is called data normalization. Suppose $\vec{x}_1, \ldots, \vec{x}_n$ are the feature vectors, let $\vec{\mu} = \frac{1}{n}\sum_{i=1}^{n}\vec{x}_i$ be their mean. Define the matrix $X = \begin{bmatrix}\vec{x}_1 & \cdots & \vec{x}_n\end{bmatrix}$, and the matrix $M = \begin{bmatrix}\vec{\mu} & \cdots & \vec{\mu}\end{bmatrix}$, where the

mean $\vec{\mu}$ appears as a column n times (M has n columns). Then, the mean normalization step which centers the data around $\vec{0}$ is the transformation $\boxed{X \mapsto \widetilde{X} = X - M}$.

Using the population convention, the covariance matrix for the centered training data is $C = \frac{1}{n}\widetilde{X}\widetilde{X}^{\mathsf{T}}$ (use $1/(n-1)$ for the sample-covariance convention). If $C = Q\Lambda Q^{\mathsf{T}}$, the stabilized whitening transformation is $\widetilde{X} \mapsto (\Lambda + \varepsilon I)^{-1/2} Q^{\mathsf{T}} \widetilde{X}$. With $\varepsilon = 0$ and positive eigenvalues, the transformed covariance is I. This is the covariance normalization step a.k.a. *whitening*.

The complete normalization transformation is $\boxed{X \mapsto (\Lambda + \varepsilon I)^{-\frac{1}{2}} Q^{\mathsf{T}}(X - M)}$, where $Q\Lambda Q^{\mathsf{T}}$ is the eigendecomposition of the feature covariance $\frac{1}{n}(X-M)(X-M)^{\mathsf{T}}$ and $\varepsilon > 0$ handles zero or very small eigenvalues. All normalization statistics must be estimated from the training set and then reused unchanged for validation and test data.

8.2 Initialization

If all the weights of the NN are initialized to zero, then neurons in the same layer can receive identical gradients and remain identical. Initializing weights too large can also saturate bounded activations. If a unit has $m = \text{fan\_in}$ independent, zero-mean, unit-variance inputs and weights drawn from $\mathcal{N}(0, \sigma^2)$, its local-field variance is approximately $m\sigma^2$. Xavier/Glorot initialization chooses variance based on fan-in and fan-out for linear, sigmoid, or tanh-style layers (commonly $2/(\text{fan\_in} + \text{fan\_out})$), whereas He initialization commonly uses $2/\text{fan\_in}$ for ReLU-style layers.

8.3 Cross-validation

Recall the bias-variance trade off and the underfitting and overfitting regimes from machine learning. A dataset from the wild is split into three parts (in some proportions):

1. Training set: Tuning model parameters
2. Validation set: Tuning hyperparameters
3. Test set: Estimating generalization error

k-fold cross-validation is a technique for hyperparameter tuning that involves splitting the dataset into k pieces, using each one as a validation set, and taking their average to "score" a chosen set of values for the hyperparameters. The hyperparameter values that show the best performance are used to train the model on the entire dataset.

8.4 Regularization

Regularization is a method of mitigating overfitting by adding a penalty term to the minimization objective, weighted by a parameter λ, such that large weights are penalized. Ridge and lasso regression techniques are examples.

8.5 Dropout

Idea: Train using only a subset of the neurons at a time. For instance, drop some neurons, along with corresponding connections, per epoch.

How does this help? It mitigates overfitting.

8.6 Expanding the Training Data

A dataset of images can be expanded by generating new samples as a result of rotations and reflections of the existing samples. In general, training data can be expanded by adding noise to the original samples.

8.7 Momentum Methods for Optimization

Momentum methods are an improvement on the GD algorithm. The idea is the introduction of a velocity parameter v, so that the updates at every time step look as follows:

$$v \leftarrow \mu v - \eta \nabla$$
$$w \leftarrow w + v\,,$$

where $\mu, \eta < 1$. This idea mitigates convergence issues caused by plummeting gradient values close to the minimum. Since the gradient values near the minimum become extremely tiny, the steps near the minimum become extremely tiny, which is an issue. The momentum methods impose that the descent have a certain "momentum" so that it actually attains the minimum. The Adam optimizer is one example of a momentum method. It is shown to converge faster than vanilla GD. Momentum methods have memory in the sense that future updates depend on past gradients. This feature proves to be beneficial. AdaGrad is one example of a method with memory that is not a momentum method. It keeps track of the magnitudes of the past gradients but involves no notion of velocity or momentum.

9 Convolutional Neural Networks

A **convolutional neural network (CNN)** is characterized by a convolutional layer which represents the operation of convolving an image with a filter. If the image has dimensions $n \times n$ and the filter has dimensions $k \times k$, the output of the convolution—called the **feature map**—will have dimensions $n - (k-1) \times n - (k-1)$. Hence, the convolutional layer

- takes n^2 inputs,
- gives $(n-k+1)^2$ outputs (has these many neurons),
- has $k^2(n-k+1)^2$ connections (i.e., it is not fully connected),
- and has k^2 learnable parameters because of *weight sharing*.

We can further parameterize a convolution operation with a *stride* s and padding p. In one spatial dimension the output size is $\left\lfloor \frac{n+2p-k}{s} \right\rfloor + 1$ (and similarly in the other dimension). Deep-learning libraries usually implement cross-correlation rather than kernel-flipped mathematical

convolution; zero padding changes output size but does not turn cross-correlation into convolution.

In a convolutional layer, the outputs are followed by a nonlinearity. This is the same as saying that we apply a nonlinearity on the feature map element-wise. Hence, an image is convolved with a filter to obtain a feature map, which is operated on element-wise by a nonlinearity. What does this achieve? It helps us perform the task of extracting simple features from the image by using different filters since a feature is represented by a feature map. The different feature maps can be thought of as different channels of a single image. If the input image has dimensions $n \times n$, and if we apply ℓ different filters, then the feature maps form a "thick" $n \times n \times \ell$ image (i.e., a $n \times n$ image with ℓ different channels). We can combine the different feature maps by convolving the "thick" image with different "thick" filters to get $n \times n$ outputs. This actually combines the features since each convolution is a weighted sum of the features followed by a nonlinearity. We can get many such $n \times n$ outputs and repeat the convolution step to get cascading convolutions, combining features from the previous layer each time.

9.1 Pooling Layers

Pooling is a technique used for dimensionality reduction in the cascading convolutions. The idea is to give up spatial resolution for efficiency.

- $r \times r$ *max pooling:* $r \times r$ block replaced by max of all r^2 entries.
- $r \times r$ *mean pooling:* $r \times r$ block replaced by arithmetic mean of all r^2 entries.

Pooling layers are typically inserted after convolutional layers to get a cascade like inputs—conv—pool—conv—pool—fc—....

9.2 Training

If two connections share the same weight, then their gradients are accumulated during training using BPA.

9.3 Residual Neural Networks

Resnets introduce skip connections across layers in the cascading pipeline of convnets. This not only improves accuracy, but also makes training more tractable. It was also found to avoid issue of vanishing/exploding gradients.

10 Unsupervised Learning

Unsupervised learning involves learning aspects/patterns in the data, or representations of the data. It does not involve predicting a label for instance. The dataset in the unsupervised learning setup consists only of the feature vectors $\mathcal{D} = \{\vec{x}_i\}_{i=1}^n$.

10.1 Clustering

Clustering is a canonical unsupervised learning problem. The task is to find *clusters* in the data, i.e. to group similar data points together to represent the data in the best possible way. Each cluster is identified by a *cluster center*. A data point would be assigned to the cluster of the cluster center which is closest to the data point in question. A diagram representing the partition of the data where each data point is mapped to one of the many clusters using the closeness metric is called a *Voronoi diagram*. The different clusters thus obtained are called *Voronoi sets*. A Voronoi diagram for all the points in the space demonstrates the quantization of space, and the Voronoi sets in this case become Voronoi regions that partition the entire space. The Voronoi regions are convex polytopes in general.

We can formulate the clustering problem as an optimization problem by coming up with an appropriate loss function. Suppose $\mathcal{C} = \{\vec{C}_1, \vec{C}_2, \ldots, \vec{C}_m\}$ are the cluster centers. Then a loss function for clustering would be

$$\ell(\mathcal{C}) = \sum_{i=1}^{n} \min_j \left\| \vec{x}_i - \vec{C}_j \right\|^2 .$$

The task then reduces to optimizing the loss function over the clusters to find the best representation; i.e., $\mathcal{C}^* = \arg\min_{\mathcal{C}} \ell(\mathcal{C})$.

$\star$ If $m = 1$, then the single best cluster center is the "center of mass" of the data.

10.1.1 k-means Algorithm for Clustering

k, here, is the number of clusters we want.

1. Initialize $\mathcal{C} = \{\vec{C}_1, \ldots, \vec{C}_k\}$.
2. **repeat until** convergence
    1. Compute the Voronoi sets $V_1, \ldots, V_k$ for $\mathcal{D}$ w.r.t. $\mathcal{C}$.
    2. Update $\vec{C}_i$ to be the center of mass of V_i: $\vec{C}_i \leftarrow \frac{1}{|V_i|} \sum_{\vec{x} \in V_i} \vec{x}$ (with a specified rule for any empty cluster).

Each assignment or centroid-update step does not increase the loss. For a finite dataset, deterministic tie handling, and a specified empty-cluster rule, the assignments eventually stabilize. The resulting fixed point is generally only a local optimum, not the global optimum.

$\star$ For a uniform distribution of points in $\mathbb{R}^2$, the global optimum of the k-means objective, in the limit of large k and away from boundaries, corresponds to a centroidal Voronoi tessellation whose cells are (asymptotically) regular hexagons.

10.1.2 Hebbian Learning

> Neurons that fire together wire together.

Hebbian learning is a NN-native algorithm invented in the 1940s. The idea is that if neuron A repeatedly excites (inhibits) neuron B, its efficiency of excitation (inhibition) increases over time. We model excitation (inhibition) by positive (negative) signals, and efficiency by

the weight. Hence, if the neuron has weight $\vec{w}$, input $\vec{x}$, and gives output y, then the update rule looks like

$$\vec{w} \leftarrow \vec{w} + \eta \vec{x} y \,,$$

where η is the learning rate. This is called the **activity product rule (APR)**. The training using the APR in this form suffers from instability because it diverges. One way to contain the divergence is by damping the positive feedback signal; i.e., to damp weight $\vec{w}$ in the APR:

$$\vec{w} \leftarrow \alpha \vec{w} + \eta \vec{x} y \,,$$

where $0 < \alpha < 1$ is a constant damping factor. A couple other variants of the APR to avoid divergence are:

- $\vec{w} \leftarrow \frac{\vec{w}+\eta\vec{x}y}{\|\vec{w}+\eta\vec{x}y\|_2}$.
- $\vec{w} \leftarrow \vec{w} + \eta(\vec{x} - \overline{\vec{x}})(y - \overline{y})$, where $\overline{\vec{x}}$ is the running average of $\vec{x}$s over time, and $\overline{y}$ is the running average of the ys over time.

10.1.3 Competitive Learning

Winner takes all.

Consider a linear layer with weights $W = \begin{bmatrix} \vec{w}_1 & \vec{w}_2 & \cdots & \vec{w}_m \end{bmatrix}$ so that it defines the input-output relationship as $\vec{y} = \varphi(W\vec{x})$. The catch is that the neurons compete for learning; i.e., only the weights of the neuron with the highest ILF are updated (only the winner gets the learning signal). Note that the output increases monotonically with the ILF; i.e., the neuron that produces the largest component of $\vec{y}$ is the winner. Hence, the update rule looks like:

$$i^* \leftarrow \arg\max_i y_i$$

$$\vec{w}_{i^*} \leftarrow \vec{w}_{i^*} + \eta(\vec{x} - \vec{w}_{i^*}) \,.$$

The logic behind the update rule becomes clearer when it's written in the form $\vec{w}_{j^*} \leftarrow (1-\eta)\vec{w}_{j^*} + \eta\vec{x}$. Hence, the new weight $\vec{w}_{j^*}$ is a weighted arithmetic mean of the old weight $\vec{w}_{j^*}$ and the input $\vec{x}$ in the form of an affine combination of the two.

11 Recurrent Neural Networks

A **recurrent neural network (RNN)** is a feedback network architecture that is used in processing sequences. The general steps involved in processing a sequence of images, for instance, using an RNN are:

1. Feed one image at a time.
2. Have each image update an internal memory.
3. Have the output for each image depend on the image as well as the memory.

This series of steps inherently defines a *recurrence*, giving RNNs their name. The internal memory of an RNN is called its state. Hence, one forward pass in an RNN takes two inputs—a data point and the state—and gives two outputs—the next (updated) state and the actual output. RNNs can be leveraged as many-to-one architectures (e.g. for sentiment analysis) as well as one-to-many architectures (e.g. for image captioning). General time series prediction tasks are many-to-one, and an RNN can be used to solve them. RNNs can also be leveraged as many-to-many architectures for tasks like machine translation (natural language translation using machine learning). Machine translation is done asynchronously; i.e., the output is generated after the entire input is read. There are tasks which require synchronous many-to-many architectures such as frame-by-frame video classification. RNNs can be leveraged for those too.

11.1 Simple RNN Architecture

Suppose $\vec{x}_t \in \mathbb{R}^d$ is the input at time step t, $\vec{s}_t \in \mathbb{R}^k$ is the state, and $\vec{y}_t \in \mathbb{R}^d$ is the output. Generally, the input and output dimensions are the same because we could need a direct connection from the output to the input. The RNN is initialized with the initial state $\vec{s}_0$. The state $\vec{s}_t$ is related to the previous state $\vec{s}_{t-1}$ and the input $\vec{x}_t$. This relationship can be described using a fully-connected neural network that looks like:

$$\vec{s}_t = \tanh(U\vec{x}_t + W\vec{s}_{t-1}),$$

where $U \in \mathbb{R}^{k\times d}$ and $W \in \mathbb{R}^{k\times k}$ are learnable parameters. This aligns with the common first intuition of combining inputs with a linear map. Next, we define

$$\vec{y}_t = \sigma(V\vec{s}_t),$$

where $V \in \mathbb{R}^{d\times k}$ is learnable.

11.2 Training and Inference

An RNN model is trained by first (conceptually) *unrolling* the recursion in the network for each training sample. This makes it a regular feed-forward network, which can be trained using the BPA. The unrolling step combined with the BPA is called the **backpropagation through time (BPTT)** algorithm.

During inference, we can unroll the recursion pertaining to a test sample to get output logits, which tell us which token is most likely the next in the sequence. It may not be desirable to always pick the most likely token since that would have very little variation. We can introduce some randomness in sampling the next tokens to increase variation during generation.

11.3 Long Short-Term Memory Networks

Standard RNNs can struggle with long-term dependencies in sequences due to vanishing and exploding gradients. **Long short-term memory (LSTM)** networks mitigate this using a memory structure that can:

1) Remember information for long periods.

2) Forget irrelevant information.

3) Control information flow through *gating* mechanisms.

LSTMs are RNNs. So suppose $\vec{x}_t$, $\vec{s}_t$, and $\vec{y}_t$ are the input, state, and output respectively at time t. An LSTM defines several gates (intermediate operations that provide the new state and the new output):

1) *Forget gate:* How much of the previous state to forget. Define

$$\vec{f}_t = \sigma(U_f \vec{x}_t + W_f \vec{y}_{t-1} + \vec{b}_f)\,,$$

where U_f, W_f, and $\vec{b}_f$ are learnable.

2) *Input gate:* How much of the *new candidate state* to keep. Define

$$\vec{i}_t = \sigma(U_i \vec{x}_t + W_i \vec{y}_{t-1} + \vec{b}_i)\,,$$

where U_i, W_i, and $\vec{b}_i$ are learnable.

3) *Output gate:* How much of the calculated output to expose. Define

$$\vec{o}_t = \sigma(U_o \vec{x}_t + W_o \vec{y}_{t-1} + \vec{b}_o)\,,$$

where U_o, W_o, and $\vec{b}_o$ are learnable.

The new candidate state is defined as

$$\vec{g}_t = \tanh(U_g \vec{x}_t + W_g \vec{y}_{t-1} + \vec{b}_g)\,,$$

where U_g, W_g, and $\vec{b}_g$ are learnable parameters. Using these definitions/operations, the new state and the new output are computed as:

$$\vec{s}_t = \vec{f}_t \otimes \vec{s}_{t-1} + \vec{i}_t \otimes \vec{g}_t$$
$$\vec{y}_t = \tanh(\vec{s}_t) \otimes \vec{o}_t$$

The state $\vec{s}_t$ is commonly called the **cell state**. The gated additive pathway can mitigate vanishing gradients and improve long-range credit assignment, but it does not guarantee the absence of vanishing or exploding gradients. In the original LSTM formulation this protected path was called the **constant error carousel (CEC)**. LSTMs are trained using BPTT.

11.3.1 Stacked LSTM

Suppose $\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_t$ is the sequence of inputs into an LSTM, then the corresponding outputs $\vec{y}_1, \vec{y}_2, \ldots, \vec{y}_t$ also form a sequence. This sequence can be passed as an input into a second LSTM for processing, which, in turn, gives a sequence of outputs $\vec{z}_1, \vec{z}_2, \ldots, \vec{z}_t$. In this way, we can create bigger and bigger stacks of LSTMs to learn more complex features in the data.

11.3.2 Bidirectional LSTM

Suppose $\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_t$ is the sequence of inputs into an LSTM for processing. We may run into a problem called the *information bottleneck* which arises from the LSTM's attempt to compress the entire context into a fixed-size vector. This may cause loss of (important) features or an incorrect weighting of the features based on significance/importance. This issue can be addressed using an architecture called the bidirectional LSTM which runs LSTM layers in both the forward and the backward directions; i.e., the forward LSTM is initialized with some initial state $\vec{s}_0$ and takes the inputs $\vec{x}_1, \ldots, \vec{x}_t$ in order, whereas the backward LSTM is initialized with a possibly different initial state $\widetilde{\vec{s}}_0$ and takes the inputs $\vec{x}_t, \ldots, \vec{x}_1$ in order. This gives two sets of outputs: the $\vec{y}_i$s from the forward LSTM and the $\widetilde{\vec{y}}_i$s from the backward LSTM. The standard way of combining the two outputs for some time $t = \tau$ is to pass $\vec{y}_\tau$ and $\widetilde{\vec{y}}_\tau$ through a (trainable) MLP to get the final output $\vec{z}_\tau$.

12 Attention

Plus LSTM.

In the domain of probabilistic modeling of language, LSTM networks run into the information bottleneck problem introduced in the subsection on bidirectional LSTMs. A much more robust solution to the information bottleneck problem is to have the model *attend to* only the relevant tokens in the context during prediction. For instance, in a machine translation task from English to French, suppose we have the sentence "Good morning ladies and gentlemen," and we want to translate it to "*Bonjour mesdames et messieurs*" in French. Then, while predicting the token "*Bonjour*", the model should only attend to the tokens "Good" and "morning" in the context. Such an *attention mechanism* can be introduced in an LSTM using an *attention layer*.

12.1 Formalism

Suppose we have an input sequence of tokens $\vec{x}_1, \ldots, \vec{x}_t$ followed by an `<EOS>`. The seq2seq LSTM algorithm computes the *encoder states* $\vec{s}_0, \ldots, \vec{s}_t$ after processing the input tokens recursively up through $\vec{x}_t$. We set the first *decoder state* $\vec{h}_1 = \vec{s}_t$, and use it to decode the first output $\vec{y}_1 = \text{LSTM}\big(\vec{h}_1, \texttt{<EOS>}\big)$. We compute the second decoder state $\vec{h}_2$ from $\vec{y}_1$ and use both $\vec{h}_2$ and $\vec{y}_1$ to compute $\vec{y}_2$. Subsequently, $\vec{y}_k = \text{LSTM}\big(\vec{h}_k, \vec{y}_{k-1}\big)$. Now we can compute the relationship between the k^{th} output token and the i^{th} input token by computing the inner product between the decoder state $\vec{h}_k$ and the encoder state $\vec{s}_i$ (there's not much else we can do with two vectors other than take their inner product). These inner product values are called the *attention scores*. For instance, the attention scores for the k^{th} decoder state are $\vec{h}_k^{\mathsf{T}}\vec{s}_1, \ldots, \vec{h}_k^{\mathsf{T}}\vec{s}_n$. There are two takeaways from this until now that apply to machine learning in general:

1. An inner product can be used as a similarity score, although its magnitude depends on vector norms and it is not a distance metric.

2. When we have scores (values) that are all over the place, it is generally a good idea to first normalize them between 0 and 1. One typical way to do it is to convert the values to a softmax distribution.

We use point (2) in our case to normalize the attention scores using a softmax distribution. Suppose the attention scores of $\vec{h}_k$ with the different encoder states are $\widetilde{\alpha}_{k1}, \ldots, \widetilde{\alpha}_{kn}$, then let $\alpha_{k1}, \ldots, \alpha_{kn}$ be their softmax-normalized counterparts. The normalized attention scores tell us the relative weights of the correlations of the decoder state with the different encoder states. For instance, in our machine translation example, we would expect a high α_{12}, a mid-range α_{11}, and low values for α_{13}, α_{14}, and α_{15}. Using the normalized attention scores, we compute the *context vector*, which is a weighted sum of the encoder states weighted by the normalized attention scores, $\vec{c}_k = \sum_{i=1}^{n} \alpha_{ki} \vec{s}_i$. The context vector is a representative vector that represents the contributions of the input tokens to the output token we want to decode. Hence, the decoding rule becomes $\vec{y}_k = \text{LSTM}\left(\vec{h}_k, \vec{y}_{k-1}, \vec{c}_k\right)$, where we combine the information in the decoder state $\vec{h}_k$ and its context $\vec{c}_k$ using a fully-connected linear layer: $A \begin{bmatrix} \vec{c}_k \\ \vec{h}_k \end{bmatrix} + \vec{b}$, where A and $\vec{b}$ are learnable parameters. This is how the attention mechanism can be added to an LSTM. Instead of a vanilla LSTM, we can use a bidirectional LSTM to calculate the encoder states as $\vec{s}_i = \begin{bmatrix} \vec{s}_i^{(f)} \\ \vec{s}_i^{(b)} \end{bmatrix}$, and stack the attention layer on top of that.

13 Transformers

> Attention is all you need.
>
> — Title of Vaswani et al. (2017)

The seminal paper by Google that introduced the transformer architecture argued that we can in fact remove the LSTM layer from "underneath" the attention layer and still achieve a good understanding of language. In other words, attention is all we need. The transformer architecture consists of two parts: the encoder and the decoder.

13.1 Encoder

The encoder architecture consists of the following stages in order:

1) Input preprocessing:
    1) Input embedding.
    2) Positional encoding.
2) N encoding blocks. One encoding block consists of:
    1) Multi-head self-attention block.
    2) Add and norm.
    3) Feed-forward NN block.

4) Add and norm.

13.1.1 Input Preprocessing

Suppose we have an input sequence of tokens $t = t_1 t_2 \dots t_n$. For instance, we are doing machine translation from French to English and our input sentence is "*bonjour mesdames et messieurs*". Then, we have input tokens t_1="*bonjour*", t_2 ="*mesdames*", t_3 ="*et*", t_4 ="*messieurs*", and t_5 = `<EOS>` (this is assuming word-level tokenization). Let $\vec{x}_i$ be the vector embedding (of some fixed dimension d) of token t_i. The embedding algorithm that maps $t_i \mapsto \vec{x}_i$ is a separate study, and we won't discuss them here. This stage that involves tokenizing the input and computing the embeddings is the *input embedding stage*.

The next stage is the *positional encoding stage*, where we add to $\vec{x}_i$ a fixed positioning vector $\vec{p}_i$ ($\in \mathbb{R}^d$) which encodes the position of the token t_i in the input sequence. For this, we make use of position embedding algorithms that map $i \mapsto \vec{p}_i$. Hence, we obtain vectors $\vec{z}_i$s such that $\vec{z}_i = \vec{x}_i + \vec{p}_i$.

13.1.2 Self-attention

In the LSTM + attention architecture, we could compute the attention scores using the encoder and decoder states we had available from the LSTM layer. In this case, we don't have such states/vectors available beforehand. Hence, to execute attention in transformers, we take motivation from KV databases. We create a synthetic database out of our $\vec{z}_i$ vectors using learnable matrices $W_K, W_V, W_Q \in \mathbb{R}^{m \times d}$, for a head dimension m that divides d: $(\vec{k}_i, \vec{v}_i) = (W_K \vec{z}_i, W_V \vec{z}_i)$ and $\vec{q}_i = W_Q \vec{z}_i$. For token t_j, we first scale the logits and then normalize them:

$$\alpha_{ji} = \frac{\exp(\vec{q}_j^{\intercal} \vec{k}_i / \sqrt{m})}{\sum_{r=1}^{n} \exp(\vec{q}_j^{\intercal} \vec{k}_r / \sqrt{m})} .$$

The factor $\sqrt{m} = \sqrt{d_k}$ controls the magnitude of the dot-product logits before softmax. Finally we compute the context vector

$$\vec{c}_j = \sum_{i=1}^{n} \alpha_{ji} \vec{v}_i \in \mathbb{R}^m .$$

Since we've created our own states/vectors from the inputs to execute attention on, this is called **self-attention**.

13.1.3 Multi-head Self-attention

To recap so far succinctly in matrix notation (the columns of the matrices are the vectors of interest): Given an input sequence $t = t_1 t_2 \dots t_n$, we compute the token embeddings $X \in \mathbb{R}^{d \times n}$ and the positional encodings $P \in \mathbb{R}^{d \times n}$ to give as output of the preprocessing stage the embeddings $Z = X + P \in \mathbb{R}^{d \times n}$.

In a self-attention layer, learnable matrices $W_K, W_V, W_Q \in \mathbb{R}^{m \times d}$ give $K = W_K Z$, $V = W_V Z$, and $Q = W_Q Z$ in $\mathbb{R}^{m \times n}$. With columns representing tokens, the row-wise attention matrix

and context are

$$A = \text{softmax}\left(\frac{Q^\mathsf{T} K}{\sqrt{m}}\right) \in \mathbb{R}^{n\times n}, \qquad C = VA^\mathsf{T} \in \mathbb{R}^{m\times n}\,.$$

This is **single-head self-attention**. In **multi-head self-attention**, each head h has its own $(W_{K_h}, W_{V_h}, W_{Q_h})$ and produces $C_h \in \mathbb{R}^{m\times n}$. If there are η heads, concatenation gives $\widetilde{C} \in \mathbb{R}^{\eta m\times n}$, and the output projection is

$$C = B\widetilde{C} + \vec{b}\vec{1}^\mathsf{T} \in \mathbb{R}^{d\times n}, \qquad B \in \mathbb{R}^{d\times \eta m}, \quad \vec{b} \in \mathbb{R}^d\,.$$

In the common choice $\eta m = d$, this projection is square and $\eta = d/m$; same-sized input and output allow MHSA blocks to be stacked.

13.1.4 Add and Norm

The add and norm is implemented as a skip connection in the encoder. The operation it describes is to add Z to C, and then normalize $C + Z$. The said normalization is a *layer normalization* operation.

Suppose $\Omega \in \mathbb{R}^{d\times n}$ has one token representation per column. Layer normalization acts across the d features independently for each token. For column j, define

$$\mu_j = \frac{1}{d}\sum_{r=1}^{d} \Omega_{rj}, \qquad \sigma_j^2 = \frac{1}{d}\sum_{r=1}^{d} (\Omega_{rj} - \mu_j)^2\,.$$

Then

$$\text{LayerNorm}(\Omega)_{rj} = \gamma_r \frac{\Omega_{rj} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}} + \beta_r\,,$$

where $\varepsilon > 0$ is numerical stabilization and $\vec{\gamma}, \vec{\beta} \in \mathbb{R}^d$ are learned scale and shift parameters. In our case, $\Omega = C + Z$. The equations below therefore describe the original transformer's post-norm convention.

13.1.5 Encoding

We have all the pieces necessary to completely describe the operation of one encoding block $\mathcal{E}$. Suppose $\mathcal{E}$ gets input Γ. Then,

$$T_\Gamma = \text{LayerNorm}(\Gamma + \text{MHSA}(\Gamma))$$
$$\mathcal{E}(\Gamma) = \text{LayerNorm}(T_\Gamma + \text{FFNN}(T_\Gamma))\,.$$

Finally, the complete operation of the encoder is $\mathcal{E}^{\circ N}(Z)$. In the original paper, $d = 512$, $N = 6$, $m = 64$, $\eta = \frac{d}{m} = 8$.

13.2 Decoder

The decoder architecture consists of the following stages in order:

1) Output preprocessing:
    1) Output embedding.
    2) Positional encoding.
2) N decoding blocks. One decoding block consists of:
    1) Masked multi-head self-attention block.
    2) Add and norm.
    3) Multi-head cross-attention block.
    4) Add and norm.
    5) Feed-forward NN block.
    6) Add and norm.
3) Linear layer.
4) Softmax.

A major difference between the encoder and the decoder is that the encoder runs only once; whereas the decoder starts with a single input embedding at time step 1, performs *autoregressive* next-token prediction, and appends that next token embedding to the original input which then becomes the new input of size $d \times 2$ at time step 2. This process continues iteratively until an `<EOS>` token is output at which point the decoder stops.

13.2.1 Output Preprocessing

At time step 1, a single initial output embedding is passed to the decoder (input is of size $d \times 1$). This initial embedding vector could be $\vec{0}$ or the embedding of a special `<BOS>` token. At time step ℓ, the initial embedding is the first column in the input of size $d \times \ell$ to the decoder. The remaining $\ell - 1$ columns are the embeddings of the tokens output at time steps 1 through $\ell - 1$. Hence, we start out with an output embedding matrix $\Xi \in \mathbb{R}^{d \times \ell}$ that is constructed as described. To this we add the positional encodings $P \in \mathbb{R}^{d \times \ell}$. Thus, we have the matrix $Y = \Xi + P \in \mathbb{R}^{d \times \ell}$ after the output preprocessing stage at time step ℓ.

13.2.2 Masked Multi-head Self-attention

Masking is necessary in the multi-head self-attention mechanism while decoding because token r cannot attend to future token positions $c > r$. With rows indexing queries and columns indexing keys, define

$$M_{rc} = \begin{cases} 0, & c \leq r \\ -\infty, & c > r \end{cases}.$$

Then $K_h, V_h, Q_h \in \mathbb{R}^{m\times\ell}$ and

$$A_h = \text{softmax}\left(\frac{1}{\sqrt{m}} Q_h^\mathsf{T} K_h + M\right) \in \mathbb{R}^{\ell\times\ell},$$

where the mask is added before the row-wise softmax. Thus $C_h = V_h A_h^\mathsf{T} \in \mathbb{R}^{m\times\ell}$; concatenation and the output projection return a matrix in $\mathbb{R}^{d\times\ell}$ when the head dimensions sum to d.

13.2.3 Multi-head Cross-attention

Multi-head cross-attention is also called *encoder-decoder attention*. If the encoder output is $E \in \mathbb{R}^{d\times n}$ and the decoder sublayer input is $S \in \mathbb{R}^{d\times\ell}$, each head projects keys and values from the encoder representations and queries from the decoder representations: $K_h = W_{K_h} E$, $V_h = W_{V_h} E$, and $Q_h = W_{Q_h} S$, all with head dimension m. We compute

$$A_h = \text{softmax}\left(\frac{1}{\sqrt{m}} Q_h^\mathsf{T} K_h\right) \in \mathbb{R}^{\ell\times n},$$

where the softmax is applied row-wise, and $D_h = V_h A_h^\mathsf{T} \in \mathbb{R}^{m\times\ell}$. Concatenating the heads and applying the output projection gives $D \in \mathbb{R}^{d\times\ell}$ when the head dimensions sum to d.

13.2.4 Decoding

Now we can piece together the operation of one decoder block $\mathcal{D}$. Suppose $\mathcal{D}$ gets input Γ. Then,

$$\begin{aligned} S_\Gamma &= \text{LayerNorm}(\Gamma + \text{MMHSA}(\Gamma)) \\ T_\Gamma &= \text{LayerNorm}(S_\Gamma + \text{MHCA}(S_\Gamma)) \\ \mathcal{D}(\Gamma) &= \text{LayerNorm}(T_\Gamma + \text{FFNN}(T_\Gamma)). \end{aligned}$$

Finally, the complete operation of the decoder to output next-token probabilities uses a vocabulary projection $W_{\text{vocab}} \in \mathbb{R}^{|\mathcal{V}|\times d}$ and bias $\vec{b}_{\text{vocab}} \in \mathbb{R}^{|\mathcal{V}|}$, followed by softmax. During training, the decoder commonly receives the ground-truth prefix (teacher forcing) while retaining the causal mask; during inference it uses the tokens generated so far.

13.3 Notes

- A standard self-attention block has $O(n^2 d)$ attention computation (alongside projection costs) and $O(n^2)$ attention-score memory per head. API pricing for input and output tokens is not explained by quadratic attention alone: autoregressive generation, the rest of the model computation, and key-value caching also matter.
- The positional encodings are sine and cosine functions of different frequencies.
- The following is a list of learnable parameters as they appear in our description of the transformer:

- W_{K_h}, W_{V_h}, W_{Q_h} for all $h \in \left\{1, \ldots, \frac{d}{m}\right\}$ in the MHSA, MMHSA, and the MHCA blocks. Hence, a total of $3 \cdot \frac{d}{m} \cdot md = 3d^2$ parameters.
- $B \in \mathbb{R}^{d \times d}$ and $\vec{b} \in \mathbb{R}^d$ in the MHSA, MMHSA, and the MHCA blocks. Hence, a total of $3d^2 + 3d = 3d(d+1)$ parameters.
- $W_{\text{vocab}} \in \mathbb{R}^{|\mathcal{V}| \times d}$ and $\vec{b}_{\text{vocab}} \in \mathbb{R}^{|\mathcal{V}|}$ in the last linear layer in the decoder, for $|\mathcal{V}|(d+1)$ parameters (possibly with weights tied to the token embeddings).
- Parameters in the FFNNs, layer-normalization scale/shift, token embeddings, and positional embeddings if learned. The items above are therefore a partial accounting per described block, not a total transformer parameter count; a total must also multiply block parameters by the number of layers.

14 Generative Pre-trained Transformer

> Language models are unsupervised multitask learners.
>
> — Title of Radford et al. (2019)

The transformer architecture we have described so far (and from the implicit description of its training), undergoes task-specific supervised learning. For instance, if the task is machine translation from French to English, then the transformer will be fed (labeled) data of French-English translation sequence pairs, where the French sequences are the feature tokens and the corresponding English sequences are the ground truth labels, akin to how an LSTM is trained. Another example of a supervised learning task is fill in the blanks, where the context phrases are the features and the filler phrases are the ground truth labels.

The original GPT work in 2018 combined transformer language-model pre-training with supervised task-specific fine-tuning; it did not invent language-model pre-training. Given a diverse corpus of unlabeled text, a decoder-only transformer can be trained to predict the next token. In current terminology this is usually called **self-supervised** learning because the prediction targets are constructed from the text itself. The generatively pre-trained language model can then be used for downstream tasks using supervised fine-tuning. Such a model is called a **generative pre-trained transformer (GPT)**. GPT uses the decoder-only causal-transformer family, not the complete encoder-decoder architecture described in the preceding section.

Simply from generative pre-training, GPT-2 exhibited zero-shot performance on several language tasks without task-specific fine-tuning. Claims that it specifically learned to write code or continue mathematical sequences are retained here only as lecture observations; they are not established results of the GPT-2 paper. **Zero-shot** prompting supplies no demonstrations of the task, **one-shot** supplies one, and **few-shot** supplies a small number. These are forms of **in-context learning** when behavior changes from context without a parameter update.

GPT-3 scaled the decoder-only GPT approach from GPT-2's largest released model of about 1.5 billion parameters to 175 billion parameters. Its in-context learning behavior was much

more pronounced. The big difference between in-context learning and fine-tuning is that fine-tuning changes model parameters through optimization, whereas in-context learning does not. **Foundation models** is the term introduced by Stanford researchers for broadly trained models adaptable to many downstream tasks.

In 2022, the InstructGPT work fine-tuned pretrained GPT-3 models using *reinforcement learning from human feedback (RLHF)*. This process involved the following steps:

1) Collect demonstration data, and train a supervised policy.
    1) A prompt is sampled from the prompt dataset.
    2) A labeler demonstrates the desired output behavior.
    3) This data is used to fine-tune GPT-3 with supervised learning.
2) Collect comparison data, and train a reward model.
    1) A prompt and several model outputs are sampled.
    2) A labeler ranks the outputs from best to worst.
    3) This data is used to train the reward model.
3) Optimize a policy against the reward model using reinforcement learning.
    1) A new prompt is sampled from the dataset.
    2) The policy generates an output.
    3) The reward model calculates a reward for the output.
    4) The reward is used to update the policy using PPO. Back to step 3.2.

This three-stage description is the InstructGPT method. ChatGPT, released on November 30, 2022, was fine-tuned from a model in the GPT-3.5 series using a related supervised-fine-tuning and RLHF process; it should not be identified simply as the InstructGPT policy described in the paper.

15 References and acknowledgments

These notes draw on explanations presented by Prof. Erdem Koyuncu in CS 559 lectures, as acknowledged in the provenance notice. Short retained lecture quotations are attributed where their speaker is known. The following concise references support the named foundational results and historical claims:

1. F. Rosenblatt, "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain," *Psychological Review*, 1958.
2. J. J. Hopfield, "Neural Networks and Physical Systems with Emergent Collective Computational Abilities," *PNAS*, 1982.
3. D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning Representations by Back-propagating Errors," *Nature*, 1986.
4. S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, 1997.
5. A. Vaswani et al., "Attention Is All You Need," 2017.
6. A. Radford et al., "Improving Language Understanding by Generative Pre-Training," 2018.
7. A. Radford et al., "Language Models are Unsupervised Multitask Learners," 2019.

8. T. Brown et al., "Language Models are Few-Shot Learners," 2020.
9. L. Ouyang et al., "Training Language Models to Follow Instructions with Human Feedback," 2022.