SoC: Learning in a Library

Reading Material

Summer'24

We introduce possibly the most influential modern machine learning innovation - **neural networks**. Imagine you're teaching a computer to recognize patterns, like recognizing whether a photo contains a cat or a dog. In traditional machine learning, we might manually define specific rules or features to help the computer make decisions. For example, we could tell it to look for specific shapes or colors.

Now, think of a neural network as an approach inspired by the human brain. Just like our brain has interconnected neurons, a neural network has layers of 'artificial neurons' or 'nodes.' These nodes work together to automatically learn patterns from the data without us having to specify explicit rules.

What's fascinating is that during training, the neural network adjusts the connections between these neurons (just like our brain strengthens or weakens connections) to get better at recognizing patterns. We show it many examples of cats and dogs, and it gets better over time at distinguishing them on its own. Note however, that analogy between NNs and the human brain stops here - the way the brain works is very different from the way neural networks learn.

Neural nets are especially powerful for tasks like image recognition, natural language processing, and many other complex problems where traditional machine learning techniques struggle. It all started in the 1940s with...

1 The McCulloch-Pitts Neuron model

This is one of the simplest models of an object that learns from data. McCulloch and Pitts proposed this model in 1943, calling it an **artificial neuron** (we will henceforth simply call it a neuron). Data $\mathbf{x} = (x_1, \ldots, x_n)$ is passed to the neuron, which has weights $\mathbf{w} = (w_1, \ldots, w_n) \in \mathbb{R}^n$, $b \in \mathbb{R}$ on its incoming edges. The neuron computes the quantity $f_{\mathbf{w},b}(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b$, where $f_{\mathbf{w},b}$ is called the aggregate function. The neuron then passes the result through a function g called the **activation function**. The output of the neuron is defined to be $g(f_{\mathbf{w},b}(\mathbf{x}))$. Note that the activation of a neuron on input $\mathbf{x}$ is simply its output on input $\mathbf{x}$.

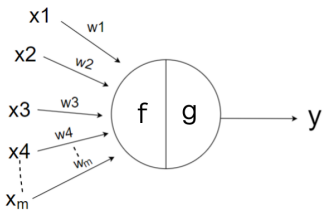


Figure 1: The McCulloch-Pitts model of an artificial neuron.

Remark. Consider the case where g is the sign function. This neuron model is precisely the perceptron model! Indeed, in 1957, Rosenblatt proposed the perceptron model inspired by the McCulloch-Pitts neuron model. And thus, one neuron can be used to perform binary classification.

2 A brief history of neural networks

- 1943: McCulloch and Pitts propose the McCulloch-Pitts neuron model.
- 1957: Rosenblatt proposes the perceptron model.

- 1960: The idea of backpropagation is proposed by a controls engineer, Kelley. It was actually developed for flight path control.
- 1969: Minsky and Papert show that the perceptron model cannot learn the XOR function.
- 1986: Rumelhart, Hinton, and Williams show that backpropagation can be used to train neural networks with hidden layers.
- 1989: The convolutional neural network is proposed by LeCun.
- 1997: The long short-term memory (LSTM) model is proposed by Hochreiter and Schmidhuber.
- 2009: Deep learning for speech recognition is proposed by Dahl.
- 2012: AlexNet wins the ImageNet competition.
- 2014: Generative adversarial networks (GANs) are proposed by Goodfellow.
- 2016: AlphaGo defeats Go world champion Lee Sedol.

Today, deep learning has taken over the machine learning community by storm by being able to solve really hard problems, often better than humans. The advent of deep (many hidden layers) neural networks has been made possible thanks to the following factors:

1. Vast amounts of data.
2. Faster computation: specialized hardware like GPUs (and recently TPUs) for matrix computation.
3. Better algorithms: better initialization, better activation functions, better optimization techniques, toolkits, libraries like TensorFlow, PyTorch, etc.

3 The Multi-Layer Perceptron

Okay, back to the neuron. The output of one neuron would serve quite well for regression where we need a scalar output (indeed, a single neuron with g being the identity is linear regression). Supposing we have a classification problem, say classify an input x into one of the classes $0, 1, \ldots, 9$? A simple way to use a network for this is to have 10 neurons, pass the same input x to all of them, and then predict the class to be the one corresponding to the neuron with the highest output value. Indeed, this is not a bad idea, and is called (multinomial) logistic regression (see this for details).

Key Idea 1
We can use outputs from neurons to make a prediction for classification/regression problems.

In general, each neuron is expected to 'do its bit', that is, capture something interesting about the data, some pattern, etc. One can think of them as mini-perceptrons, each with its own set of weights and activations, finding a separator of their inputs. Even when they receive the same input, these neurons can perceive different aspects of the data.

To illustrate, imagine two inputs that seem similar to one neuron, as their outputs are closely related, but appear quite distinct to another neuron. This diversity arises from the different weightings applied to the input in each of these neurons. Essentially, different weights capture different separations of the data.

Supposing we could use these varied outputs in some way - give them to someone to make sense of the different separations these neurons have come up with. Here is the key idea:

Key Idea 2
The outputs of neurons can be passed as input to another neuron to possibly recognize more intricate features in the separations.

Passing the output of the d neurons as the input to another neuron forms a more complicated network of interconnected neurons - the neural network. On input x, this 'second-level' or second-layer neuron now has different details about the input x fed to it by all the 'first-layer' neurons - and these new details are, crucially, non-linear. The second-layer neuron can now separate the data on non-linear boundaries. Two second-layer neurons, with their different weights, lead to two non-linear decision boundaries.

But before we revel in glory, let's see how the final output $\hat{y}$ looks like:

$$\hat{y} = g_2(\mathbf{w}^\top \mathbf{x}_1 + \mathbf{b}).$$

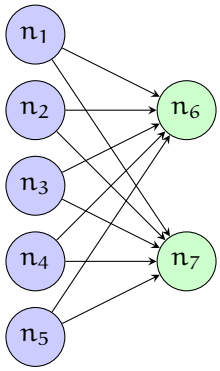


Figure 2: Connecting the output of neurons to the input of two other neurons.

Here $\mathbf{x}_1$ is the vector comprising the outputs of the first layer of neurons. That is, $(\mathbf{x}_1)_i = g_1(\mathbf{w}_i^T \mathbf{x} + b_i)$, where $\mathbf{w}_i$ is the weight vector of the i-th neuron in the first layer, and b_i is the bias of the i-th neuron in the first layer. If we suppose that the activations of the first layer are all identical (this typically does not hurt performance, but allows vectorization, which greatly speeds up training), we can write this in vectorized form as:

$$\mathbf{x}_1 = g_1(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1)$$

where $\mathbf{W}_1$ is the matrix whose i-th column is $\mathbf{w}_i$, and $\mathbf{b}_1$ is the vector whose i-th entry is b_i.
Thus, we can write:

$$\begin{aligned} \mathbf{x}_0 &= \mathbf{x} \\ \mathbf{x}_1 &= g_1(\mathbf{W}_1 \mathbf{x}_0 + \mathbf{b}_1) \\ \hat{\mathbf{y}} = \mathbf{x}_2 &= g_2(\mathbf{W}_2 \mathbf{x}_1 + \mathbf{b}_2) \end{aligned}$$

But why stop here? Perhaps we can use the outputs of the second layer to feed into a third layer, and so on. This is the idea behind **feedforward neural networks**, also called **multi-layer perceptrons** (MLPs) (the reason for the name is fairly obvious).

Key Idea 3
We can stack multiple layers of neurons next to each other to form a *deep* neural network.

4 Designing a 2 hidden layered network

This is an example of a fully-connected network, i.e., all nodes in one layer is connected to every node in the next layer. The weights in the 3 layers are of the dimensions:
The weights in the 3 layers are of the dimensions:

$$\begin{aligned} \mathbf{W}^{xp} &\in \mathbb{R}^{l \times d} \\ \mathbf{W}^{pq} &\in \mathbb{R}^{m \times l} \\ \mathbf{W}^{qy} &\in \mathbb{R}^{n \times m} \end{aligned}$$

where $\mathbf{W}_{ij}^{xp}$ refers to the weight of the connection from the i^{th} node of the input layer to the j^{th} node of the hidden layer.
Let the activation function across all the layers be $\mathbf{g}(\cdot)$.
The relation between the layers and weights can be written as:

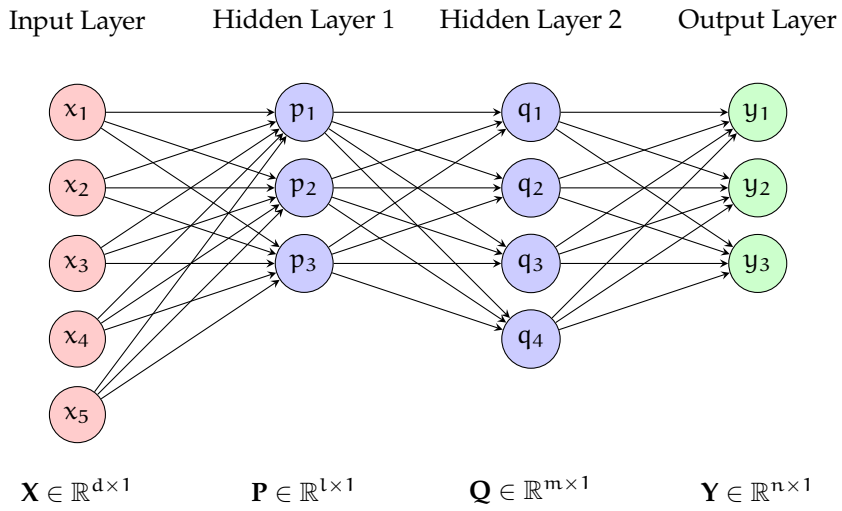


Figure 3: A (tiny) deep neural network. Note the dimensions of the input and output of each layer.

$$\begin{aligned}\mathbf{P} &= g(\mathbf{W}^{xp}\mathbf{X} + \mathbf{B}^{xp})\\ \mathbf{Q} &= g(\mathbf{W}^{pq}\mathbf{P} + \mathbf{B}^{pq})\\ \mathbf{Y} &= g(\mathbf{W}^{qy}\mathbf{Q} + \mathbf{B}^{qy})\end{aligned}$$

Here, $\mathbf{X}$ is our input, and $\mathbf{Y}$ is the corresponding classified output. $\mathbf{B}$ denotes the biases of the nodes in each layer where $\mathbf{B}^{xp} \in \mathbb{R}^{l\times 1}$, $\mathbf{B}^{pq} \in \mathbb{R}^{m\times 1}$ and $\mathbf{B}^{qy} \in \mathbb{R}^{n\times 1}$.

5 The loss function

Before we get too excited about deep neural networks, we need to figure out how to train them. That is, we need to figure out how to choose the weights and biases of the neurons so that the network does what we want it to do. To do this, as before, we need to define a loss function to tell the network how well it is doing. The loss function is a function of the weights and biases of the network, and measures how well the network is doing. The goal is to minimize the loss function.

5.1 Typical loss functions

For regression problems (the output vector of the MLP is a scalar, aka there is only one output neuron), we typically use the squared error loss. Suppose the true output is y, and the output of the MLP upon input x is $\hat{y}$. Then, the squared error loss is defined by

$$Ł(y, \hat{y}) = (y - \hat{y})^2 \implies Ł(D) = \frac{1}{N}\sum_{i=1}^{N}(y^{(i)} - \hat{y}^{(i)})^2$$

It is convex in the weights and biases and so behaves well w.r.t gradient descent.

For classification problems, the cross-entropy loss - a generalization of the binary cross-entropy from logistic regression - is typically used. Suppose we have K classes, and so the output vector of the MLP is K-dimensional. Let y be the true label of the input x, and let $\hat{y} \in \mathbb{R}^K$ be the output of the MLP on input x. The vector $\hat{y}$ represents the probability distribution over the classes predicted for input x. The cross-entropy loss is defined as:

$$L(y, \hat{y}) = -\sum_{k=1}^{K} y_k \log \hat{y}_k \implies Ł(D) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K}(y^{(i)})_k \log\left(\hat{y}^{(i)}\right)_k$$

How do we go from the neural network outputs to a probability distribution $\hat{y}_k = P(y = k|x)$ over the classes? We used the sigmoid for binary classification. Here we use a generalization, called the softmax function. The softmax function takes a K-dimensional vector z and outputs a K-dimensional vector $\sigma(z)$, where:

$$\sigma(z)_k = \frac{e^{z_k}}{\sum_{k=1}^{K} e^{z_j}}$$

6 Activation functions

This is another crucial aspect that can determine whether your model succeeds or remains stuck at a minimum. Activation functions must be chosen carefully to ensure rapid learning.

If every neuron in the network had a linear activation function, the final output would be a linear function of the input. This would essentially reduce the neural network to a single neuron with too many weights – not very useful (for an example of a linearly activated neural network that learns non-linearities for different reasons, you can refer to this).

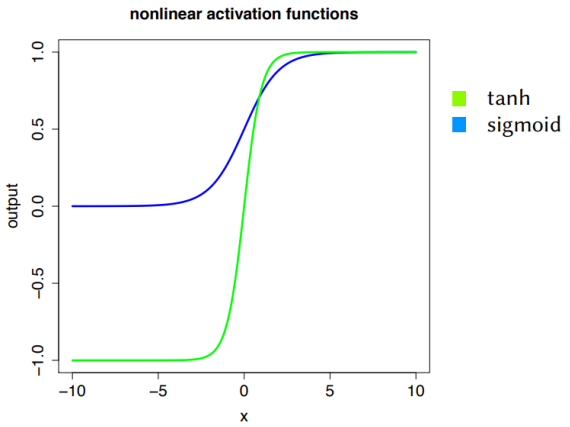
Nonlinear activation functions are where neural networks derive their power. Neural networks with just one hidden layer and a nonlinear activation can approximate *any* function with high accuracy. You can gain a deeper (visual) understanding of why this is true by checking this.

We thus need to use non-linear activation functions. Here are some common ones:

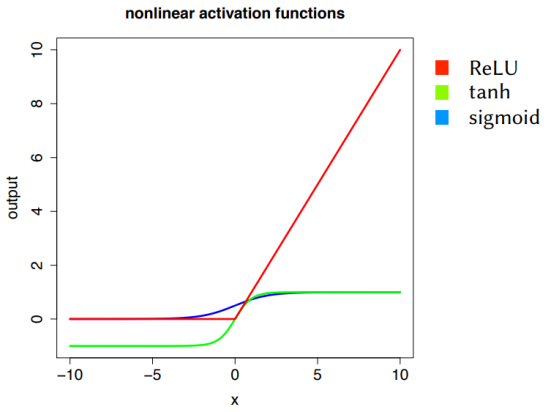
1. **Sigmoid**: $\sigma(x) = \frac{1}{1+e^{-x}}$. A common choice for the activation function. It is smooth, and bounded. However, it suffers from the vanishing gradient problem: the gradient of the sigmoid function is very small for large positive values of x, and so the weights corresponding to neurons with large activations do not get updated much. This is a problem because the neurons with large activations are the ones that are most important in the network. This slows down learning significantly. Note also that the output of the sigmoid is always positive.
2. **Hyperbolic tangent**: $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$. This is similar to the sigmoid function, but is **zero-centered**. This property helps with learning because the output values are centered around zero, which means that both positive and negative values can be learned effectively. However, like the sigmoid function, the hyperbolic tangent still suffers from the **vanishing gradient problem**.

   The hyperbolic tangent function $\tanh(x)$ can be related to the sigmoid function $\sigma(x)$ by $\tanh(x) = 2\sigma(2x) - 1$.
3. **ReLU (Rectified Linear Unit)**: The ReLU activation function is a very popular choice in neural networks due to its simplicity and the fact that it converges extremely quickly during training. It is defined as follows:

$$\text{ReLU}(x) = \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases}$$



((a)) tanh and sigmoid suffer from the vanishing gradient problem.



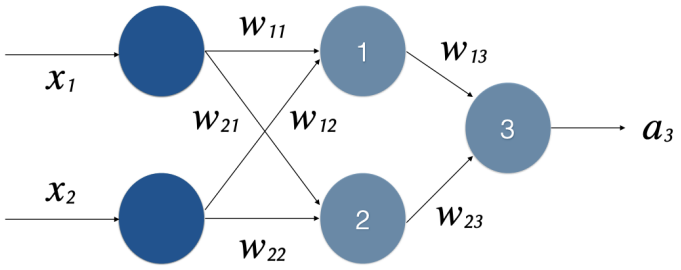
((b)) The three activations together.

4. **Many more!**: Leaky ReLU, ELU, SELU, etc.

7 Feed-forward Neural Networks

These are quintessentially deep learning models because they involve learning the parameters to accurately match the output. The feed-forward in the name comes from the fact that in computing the output of the network, information only flows in one direction without any feedback connections. In case such feedback computations are a part of the models, they are referred to as **Recurrent Neural Networks (RNNs)**.
The structure of a neural network consists of a number of *hidden* layers, each representing a vector value. Each value of the vector represents a *neuron* which acts as a vector-to-scalar function. These neuron like units of each layer work in parallel, computing their own outputs using the values of the previous layer. The ideas of having multiple layers, with each unit having an activation function are drawn from neuroscience.

7.1 A simple example



Consider the above network which computes the output a_3. For the first hidden layer [Nodes 1, 2], the weighted input values are:

$$\begin{aligned} z_1 &= w_{11}x_1 + w_{12}x_2 + b_1 \\ z_2 &= w_{21}x_1 + w_{22}x_2 + b_2 \\ \implies \boldsymbol{z}^1 = \begin{pmatrix} z_1 & z_2 \end{pmatrix} &= \begin{pmatrix} x_1 & x_2 \end{pmatrix} \begin{pmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \end{pmatrix} + \begin{pmatrix} b_1 & b_2 \end{pmatrix} \\ \boldsymbol{z}^1 &= \boldsymbol{x}W^1 + \boldsymbol{b}^1 \end{aligned}$$

The superscript here denotes the layer number. These units apply a non-linear activation function g on these weighted sums, giving:

$$\boldsymbol{a}^1 = \begin{pmatrix} a_1 & a_2 \end{pmatrix} = \begin{pmatrix} g(z_1) & g(z_2) \end{pmatrix} = g(\boldsymbol{z}^1)$$

A similar computation happens in going from the hidden layer to the output layer [Node 3]:

$$\begin{aligned} z_3 &= w_{13}a_1 + w_{23}a_2 + b_3 \\ &= \begin{pmatrix} a_1 & a_2 \end{pmatrix} \begin{pmatrix} w_{13} \\ w_{23} \end{pmatrix} + b_3 \\ \implies \boldsymbol{z}^2 &= \boldsymbol{a}^1 W^2 + \boldsymbol{b}^2 \\ a_3 &= g(z_3) \\ \implies \boldsymbol{a}^2 &= g(\boldsymbol{z}^2) \end{aligned}$$

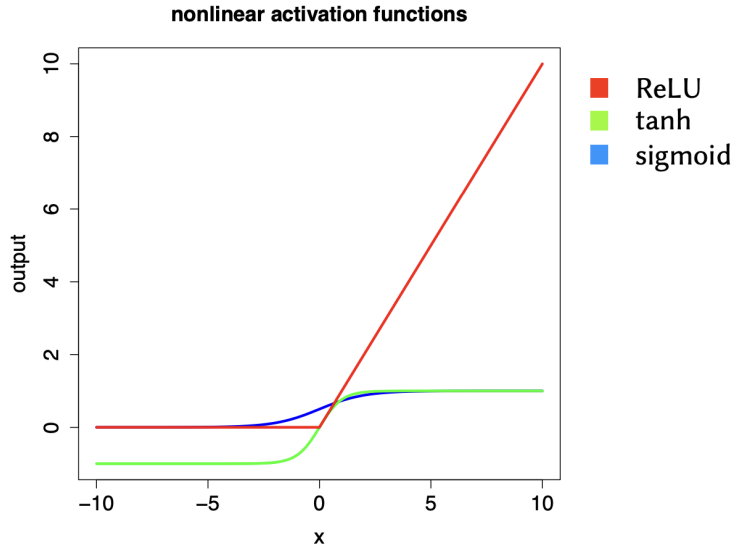
Combining these, we can write:

$$a_3 = g(g(\boldsymbol{x}W^1 + \boldsymbol{b}^1)W^2 + \boldsymbol{b}^2)$$

7.2 Activation functions

In general, a wide variety of differentiable functions perform perfectly well as activation functions. Some commonly used ones are:

- **Sigmoid:** $\sigma(x) = \dfrac{1}{1+e^{-x}}$
- **Hyperbolic tangent(tanh):** $\tanh(x) = \dfrac{e^{2x}-1}{e^{2x}+1}$
- **Rectified Linear Unit (ReLU):** $\mathrm{ReLU}(x) = \max(0,x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$



The **Rectified Linear Unit (ReLU)** activation function is easy to implement and offers better performance and generalization in deep learning as compared to the sigmoid and tanh activation functions. ReLU represents a nearly linear function and therefore preserves the properties of linear models that makes them easy to optimize with gradient descent methods. This function eliminates the vanishing gradient problem for large inputs, as observed in the other activation functions

8 Training Neural Networks

8.1 Loss function

The aim of training a NN is to minimize the loss:

$$J(\theta) = \sum_{(\boldsymbol{x_i}, y_i) \in \mathcal{D}_{\text{train}}} \mathcal{L}(\mathrm{NN}(\boldsymbol{x_i}; \theta), y_i)$$

where θ stands for all the weights, biases $(W^l, \boldsymbol{b}^l)$ of all the layers in the network.
A popular choice for $\mathcal{L}$ in case of binary labelled data is the binary cross entropy loss:

$$J(\theta) = \sum_{(\boldsymbol{x_i}, y_i) \in \mathcal{D}_{\text{train}}} y_i \log(\mathrm{NN}(\boldsymbol{x_i}; \theta)) + (1 - y_i) \log(1 - \mathrm{NN}(\boldsymbol{x_i}; \theta))$$

8.2 Gradient Descent

Stochastic Gradient Descent (SGD):

Inputs: $NN(x;\theta)$, Training examples $x_1, \ldots, x_n$; Outputs, $y_1, \ldots, y_n$
Loss function: $L(NN(x_i;\theta), y_i)$
  Randomly initialize θ
  do until **stopping criterion**
    Pick a training example (x_i, y_i)
    Compute the loss $L(NN(x_i, \theta), y_i)$
    Compute the gradient of L, $\nabla_\theta L$ with respect to θ:
    Update θ: $\theta \leftarrow \theta - \eta \nabla_\theta L$, where η is Learning rate
  done
Return: θ

Mini-batch Gradient Descent (GD):

Inputs: $NN(x;\theta)$, Training examples $x_1, \ldots, x_n$; Outputs, $y_1, \ldots, y_n$
Loss function: $L(NN(x_i;\theta), y_i)$
  Randomly initialize θ
  do until **stopping criterion**
    Randomly sample a batch of training examples $\{x_i, y_i\}_{i=1}^{b}$
      (where the batch size, b, is a hyperparameter)
    Compute the gradient of L over the batch, $\nabla_\theta L$ with respect to θ:
    Update θ: $\theta \leftarrow \theta - \eta \nabla_\theta L$, where η is Learning rate
  done
Return: θ

For this, we need to find $\frac{\partial L}{\partial w}$ for every weight w, and update it as

$$w \leftarrow w - \eta \frac{\partial L}{\partial w}$$

We can efficiently compute $\frac{\partial L}{\partial w}$ using Back-Propagation algorithm.

9 Back Propagation

Training a feed-forward neural network hinges on the problem of computing the gradient of the loss function with respect to the parameters efficiently. The back propagation algorithm applies the chain rule of calculus in a specific order to accomplish this in general for any *computational graph*. First, a simple introduction to back propagation is provided, followed by a more generalised study.

9.1 Back Propagation: An overview

We need to find $\frac{\partial L}{\partial w}$ for every weight w. We will do this by first finding $\frac{\partial L}{\partial u}$ for every node u and use following to find $\frac{\partial L}{\partial w}$:

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial u} \cdot \frac{\partial u}{\partial w}$$

To efficiently compute $\frac{\partial L}{\partial u}$ for every node u, we will use **Chain rule of differentiation** recursively for every layer. If L can be written as a function of variables v_1, ..., v_n, which in turn depend on another variable u, then

$$\frac{\partial L}{\partial u} = \sum_i \frac{\partial L}{\partial v_i} \cdot \frac{\partial v_i}{\partial u}$$

Let $\Gamma(u) = \{v_1, ..., v_n\}$ represent the children of u, then the chain rule gives

$$\frac{\partial L}{\partial u} = \sum_{v \in \Gamma(u)} \frac{\partial L}{\partial v_i} \cdot \frac{\partial v_i}{\partial u}$$

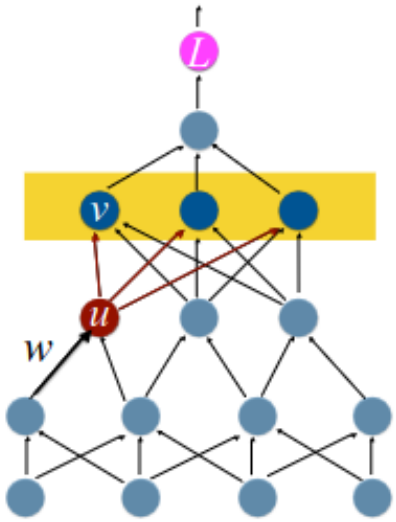


Figure 5: Image of a typical graph

In forward pass, we compute and store the activation function values for each node, to be used later in finding derivatives in backpropagation.

9.2 Back Propagation: A detailed study

Generalized chain rule

Let $g : \mathbb{R}^m \to \mathbb{R}^n$ and $f : \mathbb{R}^n \to \mathbb{R}$ be two maps. If $z = f(\boldsymbol{y}), \boldsymbol{y} = g(\boldsymbol{x})$ where $\boldsymbol{y} \in \mathbb{R}^n, \boldsymbol{x} \in \mathbb{R}^m$ then:

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}$$

$$\nabla_{\boldsymbol{x}} z = \begin{pmatrix} \vdots \\ \frac{\partial z}{\partial x_i} \\ \vdots \end{pmatrix} = \begin{pmatrix} & \vdots & \\ \frac{\partial y_1}{\partial x_i} & \cdots & \frac{\partial y_n}{\partial x_i} \\ & \vdots & \end{pmatrix} \begin{pmatrix} \frac{\partial z}{\partial y_1} \\ \vdots \\ \frac{\partial z}{\partial y_n} \end{pmatrix}$$

$$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^\top \nabla_{\boldsymbol{y}} z$$

where $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ is the $n \times m$ Jacobian matrix of g.

Backprop for feed-forward NNs

Consider that for a single datapoint $(\boldsymbol{x}, y)$ we aim to compute the gradients of $\mathcal{L}(\text{NN}(\boldsymbol{x}; \theta), y)$. Let the NN have L layers, with the lth layer having n^l nodes, whose input is $\boldsymbol{z}^l$, and whose non-linear output is $\boldsymbol{a}^l = f^l(\boldsymbol{z}^l)$.
Based on the model seen, let $\boldsymbol{z}^l = W^l \boldsymbol{a}^{l-1} + \boldsymbol{b}^l$, and $\boldsymbol{a}^0 = \boldsymbol{x}$, $a^L = \text{NN}(\boldsymbol{x}; \theta)$.
We can now use the chain rule to compute the gradient of the loss wrt to some layer's input given the same

for the next layer.

$$\nabla_{z^l}\mathcal{L} = \left(\frac{\partial z^{l+1}}{\partial z^l}\right)^\top \nabla_{z^{l+1}}\mathcal{L}$$

Considering

$$\begin{aligned}\frac{\partial z_i^{l+1}}{\partial z_j^l} &= \frac{\partial (W^{l+1}a^l + b^l)_i}{\partial z_j^l}\\ (W^{l+1}a^l + b^l)_i &= \sum_k W_{ik}^{l+1} f^l(z_k^l) + b_i^l\\ \implies \frac{\partial z_i^{l+1}}{\partial z_j^l} &= W_{ij}^{l+1}\cdot (f^l)'(z_j^l)\end{aligned}$$

Hence,

$$\frac{\partial z^{l+1}}{\partial z^l} = W^{l+1}\cdot (f^l)'(z^l)$$

where $(f^l)'(z^l)$ is $\mathrm{diag}((f^l)'(z_1^l),\ldots,(f^l)'(z_{n^l}^l))$, which finally gives us:

$$\nabla_{z^l}\mathcal{L} = (f^l)'(z^l)\cdot (W^{l+1})^\top \cdot \nabla_{z^{l+1}}\mathcal{L}$$

Moreover, at the last layer [assumed to have a single node]:

$$\begin{aligned}\nabla_{z^L}\mathcal{L}(\mathrm{NN}(x;\theta), y) &= \nabla_{z^L}\mathcal{L}(f^L(z^L), y)\\ &= \left.\frac{\partial \mathcal{L}(\hat{y}, y)}{\partial \hat{y}}\right|_{\hat{y}=\mathrm{NN}(x;\theta)} \cdot (f^L)'(z^L)\end{aligned}$$

Now we can utilize these gradients to compute the gradients of the loss for the parameters. Consider any weight W_{ij}^l, it contributes to z_i^{l+1} weighted with a factor of a_j^l:

$$\begin{aligned}\implies \frac{\partial \mathcal{L}}{\partial W_{ij}^l} &= \frac{\partial \mathcal{L}}{\partial z_i^{l+1}} a_j^l\\ \nabla_{W^l}\mathcal{L} &= \nabla_{z^{l+1}}\mathcal{L}\cdot (a^l)^\top\end{aligned}$$

Similarly for the biases, they contribute with a factor of 1 in the same manner as above:

$$\begin{aligned}\implies \frac{\partial \mathcal{L}}{\partial b_i^l} &= \frac{\partial \mathcal{L}}{\partial z_i^{l+1}}\\ \nabla_{b^l}\mathcal{L} &= \nabla_{z^{l+1}}\mathcal{L}\end{aligned}$$

9.3 The algorithm

As seen above, the computation of the gradient can be done in a backwards manner, but it relies on both of the a^l and z^l values at each layer. These are computed in a forward pass.
Once we have the necessary gradients, the next step is gradient descent:

$$\theta \leftarrow \theta - \eta\nabla_\theta L$$

If the gradient is computed over a randomly sampled (x_i, y_i) from the training dataset, it is referred to as **Stochastic Gradient Descent (SGD)**.
Otherwise, if the gradient is computed over a randomly sampled *batch* $\{(x_i, y_i)\}_{i=1}^b$, it is called **Mini-batch Gradient Descent**.

Algorithm 1: Back propagation

Assume that the nodes of the network are $u_1, u_2, \ldots, u_n = L$ in a topologically sorted manner, where L is the unit computing the final loss function;
Forward pass;
for $i \leftarrow 1$ **to** n **do**
    $z(u_i) = \sum_{j:u_j \in \text{Parents}(u_i)} w_{ij}\, a(u_j)$;
    $a(u_i) = g(z(u_i))$
Backward pass;
for $i \leftarrow n$ **to** 1 **do**
    $\partial L/\partial z(u_i) = \sum_{j:u_j \in \text{Children}(u_i)} \partial L/\partial z(u_j) * \partial z(u_j)/\partial z(u_i)$;
for w_{ij} **do**
    w_{ij} feeds into node i;
    $\partial L/\partial w_{ij} = \partial L/\partial z(u_i) * \partial z(u_i)/\partial w_{ij}$;

10 Regularization

In the context of deep learning, most regularization strategies are based on regularizing estimators. Regularization of an estimator works by trading increased bias for reduced variance. Some popular regularization techniques in deep learning:
Let us skip upon the pre-Deep Learning Era regularizations such as **Ridge & Lasso** regularizations.

10.1 Label Smoothing

Noise injection is one of the most powerful regularization strategies. **By adding randomness, we can reduce the variance of the models and lower the generalization error**. The question is how and where do we inject noise?

Label smoothing is a way of adding noise at the output targets, aka labels. Let's assume that we have a classification problem. In most of them, we use a form of cross-entropy loss such as $-\sum_{j=1}^{c} y_{i,j} \cdot \log(P_{i,j})$ and softmax function to output the final probabilities of labels.

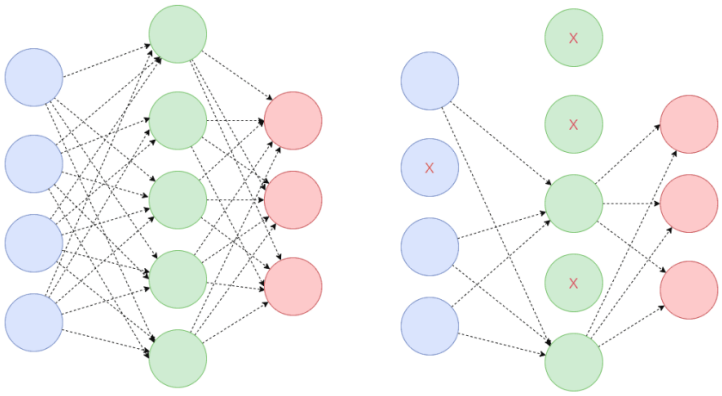
If the target vector is of the form $\{0, 1, 0, 0\}$. Because of the way softmax is defined, it can never achieve a value of 0 or 1. Hence it continues to train on the same dataset improving upto the output probabilities to reach extremes of 0 and 1 as a result leans to the side of "overfitting" on the training data.

To address this issue, label smoothing introduces a small margin ϵ over the hard 0 and 1 in the output vectors. Specifically 0 is replaced by $\frac{\epsilon}{1-c}$ and 1 by 1-ϵ where c is number of classes.

10.2 Dropout

Dropout falls into noise injection techniques and can be seen as **noise injection into the hidden units of the network.** During training, some neurons of hidden layers are randomly dropped out for a forward pass with probability p
.
During test time though, all units will be present but their outputs will be scaled down by a factor of p. This is done to achieve the model similar to one we used in training.

By using dropout, the same layer will alter its connectivity and will search for alternative paths to convey the information to the next layer. As a result, each update to a layer during training is performed with a different "view" of the configured layer.
Conceptually, it approximates training a large number of neural networks with different architectures in parallel.



Dropout has the effect of making the training process noisy.This conceptualization suggests that perhaps dropout breaks up situations where network layers co-adapt to correct mistakes from prior layers, making the model more robust. It increases the sparsity of the network and in general, encourages sparse representations!

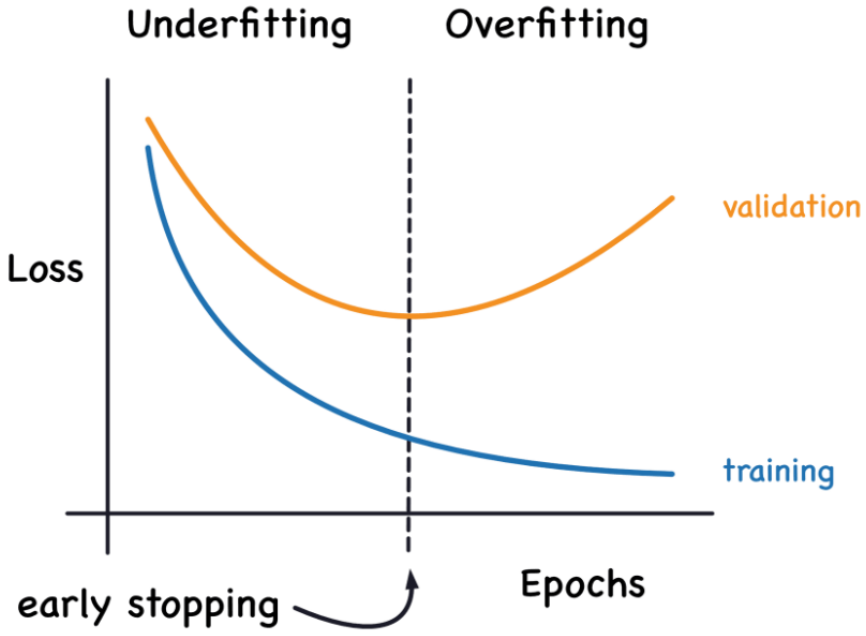
10.3 Stochastic Depth

Stochastic step goes a step further and drops entire network layers while keeping them intact during testing.In particular, it drops layers that have *Residual connections* with a probability p that is a function of **depth** of that layer.

10.4 Early Stopping

Perhaps the most commonly used regularization technique due to it's simplicity and effectiveness.It refers to the process of **stopping training when validation error is no longer decreasing.**

A different way to think of early stopping is as a very efficient hyperparameter selection algorithm, which sets the number of epochs to the absolute best. It essentially restricts the optimization procedure to a small volume of the trainable parameters space close to the initial parameters.



11 Scheduling Learning Rates

As you have guessed, it is trivial in employing same learning rate to all the features of inputs. This however need not capture all the properties of different features. We want learning rate to be responsive in each feature and time and hence several methods are employed in scheduling learning rates. Some examples are:

- Step Decay
- Exponential Decay
- Adagrad
- Adam

Adagrad and *Adam* are adaptive learning rate methods in the sense that they tune learning rates independently for different features wrt time.

12 Backpropagation: Recap

Backpropagation is a method used to find the gradients in a computational graph. The computational graph is constructed in such a way that it is a directed acyclic graph. Let us say that we have nodes in the computational graph $u, v_1, v_2, \cdots v_n$ as shown in the diagram, in such a way that there are edges from u to v_i $\forall i \in \{1, 2, \cdots n\}$ and these are the only edges emanating from u. Hence, the node u is "dependent" on nodes $v_1, v_2, \cdots v_n$ for backpropagation (whereas in the forward propagation, the nodes $v_1, v_2, \cdots v_n$ are dependent on u). Let us say that the gradients of the loss function $\mathcal{L}$ with respect to v_i's are

$$\frac{\partial \mathcal{L}}{\partial v_i}$$

Then, the gradient of the loss function with respect to u can be written as

$$\frac{\partial \mathcal{L}}{\partial u} = \sum_{i=1}^{n} \frac{\partial \mathcal{L}}{\partial v_i} \cdot \frac{\partial v_i}{\partial u}$$

This is basically the chain rule for partial derivatives. To utilize this equation for backpropagation, we would assume that in the above equation, $\frac{\partial \mathcal{L}}{\partial v_i}$ $\forall i$ are known before computing the gradient with respect to u, and since there is an edge from u to v_i, there is a function f_i such that $v_i = f_i(u)$(keeping other variables and parameters fixed). Hence, the second terms in the summation, $\frac{\partial v_i}{\partial u}$, are also easily computable.

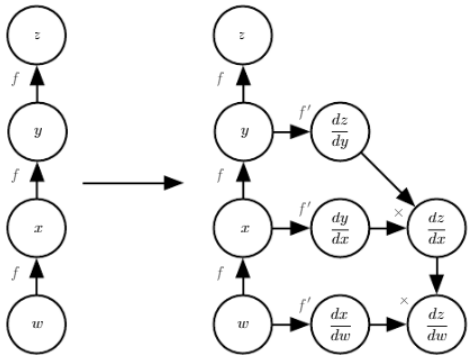


Figure 6: This diagram is taken from *Deep Learning by Goodfellow, Bengio and Courville*. It depicts backpropagation for a linear computational graph.

The above equation can be applied recursively over the layers in a neural network, and since the structure of the computational graph is a directed acyclic graph, this recursion will eventually terminate on reaching the input layer, or the layer immediately next to it. The base case would be initializing the gradient of the node in the computational graph corresponding to the loss function $\mathcal{L}$ to be 1, since the gradient of $\mathcal{L}$ with respect to itself is 1.

13 Regularization

13.1 L_2 Regularization

This regularizer has been our friend since the time we started learning regularization in Machine Learning. Consider the case of multi-class classification in a neural network. The final layer in the network would typically be a softmax layer which converts real-valued outputs to probabilites - numbers in the range $(0, 1)$. Let $\mathcal{D} = \{(x_i, y_i)\}_{i \in [N]}$ be the training data where x_i's are the features and y_i's are the corresponding class labels. Let $\{\hat{y}_{ik}\}$ be the softmax output of input x_i corresponding to the k^{th} class as computed by the neural

network. Considering the weights of the neural network concatenated in a single flat vector $\mathbf{w}$, we can define the regularized cross entropy loss as

$$\mathcal{L}_{\mathbf{w}}(\mathcal{D}) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K}\mathcal{I}(y_i = k)\log \hat{y}_{ik} + \beta \mathbf{w}^T\mathbf{w}$$

where $\mathcal{I}$ is the indicator function and β is the regularization strength. Note that there may be other forms of L_2 regularization as well which assign different regularization strengths to weights corresponding to different layers of the network. Biases aren't included in $\mathbf{w}$ because it can lead to underfitting, just as was the case in regularized linear regression.

13.2 Dropout Regularization

This kind of regularization is native to neural networks. It is simple to implement and has proven to be quite effective in practice. To motivate this regularizer, consider *bagging* in classical ML. In essence, an ensemble of machine learning models are trained over different training sets and then each test point is evaluated using multiple trained models. This is not easy to achieve in deep learning because the training time for deep networks is high and training itself is memory intensive. *Dropout* provides a simple surrogate to this problem. The general intuition is that we want to create(or pretend to create) an ensemble of neural networks by removing one or more neurons from the base architecture. Consider a number $p \in (0, 1)$. The training algorithm can be written as follows

1. For every batch $\mathcal{B}$ of the training data, do the following
2. Consider the base neural network, now for each neuron that is **not** in the input layer or the output layer, choose to deactivate that neuron i.e. make its output zero with a probability p.
3. Train this subset of the neural network on $\mathcal{B}$.

Note that unlike bagging, there is *parameter sharing* in dropout regularization. That is, the parameters from a previous training step are carried over to the next training step, which uses a different subset of the neural network whereas in bagging, each model is trained independently of the others. This difference between dropout and bagging allows dropout to be an an inexpensive surrogate for bagging. There have been studies where p changes across layers of the network, which did not lead to great improvements. During forward propagation, dropout requires that the neurons which have been deactivated in that training step must have output zero. This is fairly easy to implement - just use a mask vector for each layer and multiply the layer output with mask. During backpropagation with dropouts, the values of the nodes post-dropout must be used in the computational graph, and not pre-dropout. This essentially just replaces the dropped nodes with zero without having to modify the algorithm for backpropagation.

13.2.1 During Testing

There are two methods to evaluate the neural ensemble on a test datum:

- **Method 1**: Create multiple sub-networks using the same dropout probability p and average the predictions from all these sub-networks. This is similar to evaluation method in bagging.
- **Method 2**: There is another way to evaluate the test datum which uses a single forward propagation of the entire base network(without any dropout), and perform well empirically. Intuitively, each neuron will be active with a probability $1 - p$ in the ensemble. So, the idea is to multiply the weights attached to each neuron by $1 - p$ and making one single forward propagation to get the result corresponding to the test datum.

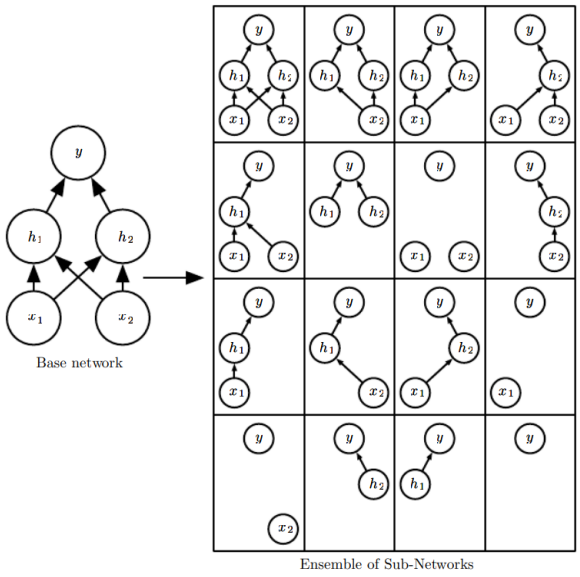


Figure 7: This diagram is taken from *Deep Learning by Goodfellow, Bengio and Courville*. It shows the various possible sub-networks that can be formed using dropouts.

13.3 Early Stopping

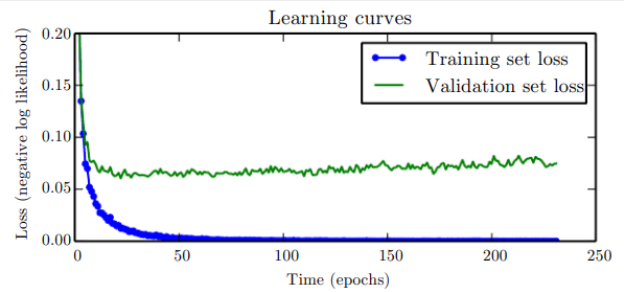


Figure 8: This diagram is taken from *Deep Learning by Goodfellow, Bengio and Courville*.

Let us recall the above graph of training set and validation set losses vs number of epochs. Observe that the training objective decreases consistently over time, but the validation set average loss eventually begins to increase again, forming an asymmetric U-shaped curve.
Ideally, one would want training to proceed until the point he/she is satisfied with the validation and training accuracies/loss values achieved. That is, a point where the training loss is not too low implying a higher validation loss neither too high which again implies higher validation loss. One way to implement this is to keep a track of the number of consecutive epochs where the validation loss has failed to decrease, keep storing the models with the best intermediate validation error and stop training after the validation error has failed to decrease over a certain number of epochs(this is a hyperparameter and needs to be tuned) i.e. *early stopping*.
Early stopping can also be used along with other regularizers, since it is easy to implement without changing the learning step. For neural networks especially, it turns out that *early stopping* works well in that it is believed that neural networks tend to remember training examples as the number of epochs becomes high.

13.3.1 Additional Reading - Double descent

Double descent is a phenomenon where we observe strong test performance from very overfit, complex models. As we can see in Figure 9, the test error first decreases as the model complexity increases, then

increases because of overfitting, and then decreases again as the model complexity increases further (i.e. the model is even more overfit).
It seems to suggest that memorization eventually leads to better generalization.

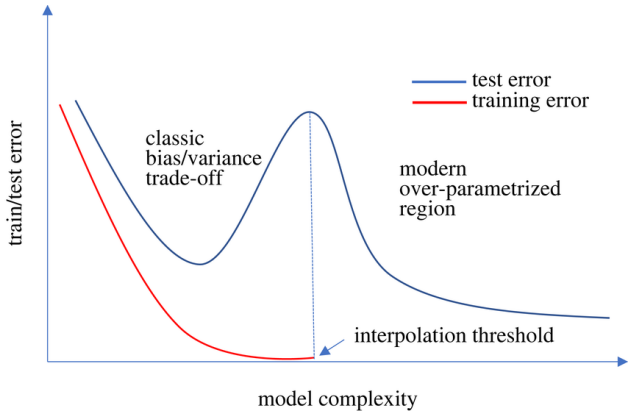


Figure 9: Double Descent

14 Learning Rate Scheduling

The fundamental gradient descent algorithm i.e.

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \mathcal{L}_{\mathbf{w}}(\mathcal{D})$$

η being the learning rate, assumes a constant learning rate throughout the training process. In practice it has been observed that the convergence is better on decaying the learning rate, either in a linear way or an exponential way. This can be understood in the sense that we do not want to take long steps as we move closer to the minima. The following graph shows training accuracy vs learing rate.

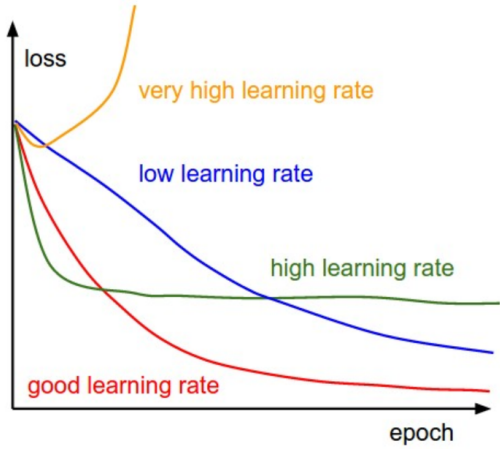


Figure 10: This diagram is taken from CS231n of Stanford University

As can be seen, high learning rates initially move faster but then do not get to the minima whereas low learning rates are too slow to converge. Very high learning rates can even lead to divergence. In between all these is an optimal learning rate schedule which moves at the right speed to the right minima. Adaptive optimizers such as **Adagrad** and **Adam** perform online learning rate scheduling by examining the gradients, with Adam being one of the best optimizers.
As a sidenote, **Simulated Annealing** on learning-rate can be used alongside gradient descent to act as a learning-rate scheduler.
In the next section, we will look at some popular optimizers and understand the concept of **momentum**.

15 Optimizers

Notation: Henceforth, $\mathbf{g}_t$ will denote the expression $\nabla_{\mathbf{w}} \mathcal{L}_{\mathbf{w}_{t-1}}(\mathcal{D})$

Let us first look at the drawbacks of vanilla gradient descent algorithm.

- In complex scenarios, there can be situations with high gradients which induces a high variance in the weight parameters and the loss function. This is detrimental to achieving the goal of the descent algorithm.
- If there are *plateau* regions, i.e. flat regions then the gradient in those regions will be close to zero and hence making no/very slow progress on descent. This can potentially occur when there are multiple undesirable local minima.

We will now make a "first" improvement to the vanilla SGD algorithm by using *momentum*.

15.1 SGD + Momentum Optimizer

Consider a number $\beta \in [0, 1]$. Define velocity terms $\mathbf{v}_t$ for each training step t as follows

$$\mathbf{v}_t \leftarrow \beta \mathbf{v}_{t-1} + \eta \mathbf{g}_t \tag{1}$$

Redefine the weight update step as

$$\mathbf{w}_t \leftarrow \mathbf{w}_{t-1} - \mathbf{v}_t \tag{2}$$

To get an understanding of what velocity is actually contributing to, let us expand the RHS of the velocity update

$$\begin{aligned}
\mathbf{v}_t &= \eta \mathbf{g}_t + \beta \mathbf{v}_{t-1} \\
\mathbf{v}_t &= \eta \mathbf{g}_t + \beta(\eta \mathbf{g}_{t-1} + \beta \mathbf{v}_{t-2}) \\
&\vdots \\
\mathbf{v}_t &= \eta \mathbf{g}_t + \beta \eta \mathbf{g}_{t-1} + \beta^2 \eta \mathbf{g}_{t-2} + \cdots + \beta^t \mathbf{v}_0
\end{aligned}$$

If we set $\mathbf{v}_0$ to 0, then the velocity is just an exponentially weighted sum of all the gradients calculated till that point. Hence, it *smoothens* gradients and will aid in making movement even when there are flat regions. However, there is a chance of overshooting local minimas because velocity includes a weighted sum of gradients, which may not be zero at a local minima of the loss function.

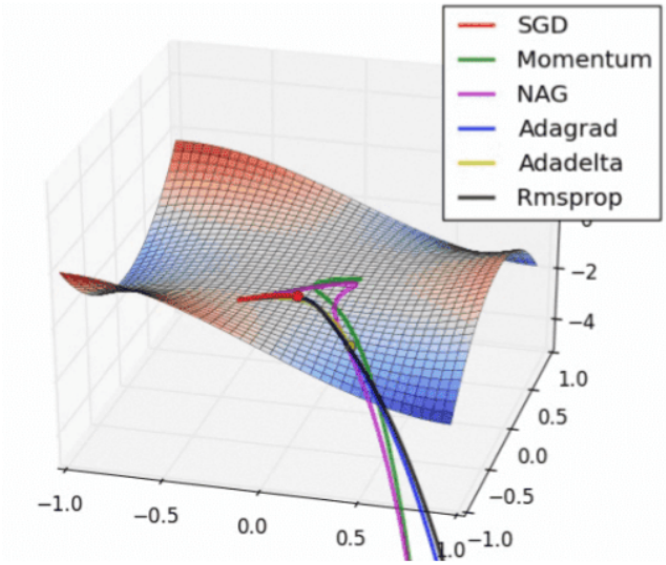


Figure 11: This diagram is by Alec Radford. It shows the paths taken by different optimizers during gradient descent.

15.2 Adagrad Optimizer

Adagrad is an online learning rate scheduler. It is also one of those few examples in machine learning where theoretical studies and empirical studies reinforce each other. Consider two weight parameters $\mathbf{w}_1$ and $\mathbf{w}_2$ such that

$$\nabla_{\mathbf{w}_1} \mathcal{L}_{\mathbf{w}}(\mathcal{D}) >> \nabla_{\mathbf{w}_2} \mathcal{L}_{\mathbf{w}}(\mathcal{D})$$

In such a case, using the same small learing rate for both $\mathbf{w}_1$ and $\mathbf{w}_2$ will cause a significant change in $\mathbf{w}_1$ and little or no change to $\mathbf{w}_2$. This is undesirable as large weight updates might be detrimental to the descent's progress. Hence, there is a need to have learning rates that adapt to parameters. On this note, let us define $\mathbf{s}_t$ for each training step such that

$$\mathbf{s}_t = \mathbf{s}_{t-1} + \mathbf{g}_t \odot \mathbf{g}_t \tag{3}$$

where $\odot$ denotes element-wise product. This way, if the gradients $\mathbf{g}_t$ are large, then $\mathbf{s}_t$ will be large. The weight update step will now be

$$\mathbf{w}_t \leftarrow \mathbf{w}_{t-1} - \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \odot \mathbf{g}_t \tag{4}$$

ϵ is to prevent numerical overflows during the initial epochs if s_0 is set to 0.
Observe that the effective learning rate will be

$$\eta' = \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}}$$

That is, the parameters with high values of gradient will have low effective learning rate and the parameters with lower values of gradient will have higher effective learning rate. This is exactly what we desired. However, $\mathbf{s}_t$ monotonically increases with t, so the effective learning rate will decrease with t and it might be very slow to converge after a point. RMSProp Optimizer deals with this problem. Also, it has been observed that removing the square-root from the effective learning rate degrades the performance of the optimizer. Can you think of why is that so?

15.3 RMSProp Optimizer

This makes just one change to Adagrad, which is the following

$$\mathbf{s}_t = \gamma \mathbf{s}_{t-1} + (1-\gamma)\mathbf{g}_t \odot \mathbf{g}_t \tag{5}$$

where $\gamma \in [0, 1]$
Now, no longer does $\mathbf{s}_t$ monotonically increase with t. In fact, it relies more on the recent gradients, just like velocity. If the recent gradients are small, then $\mathbf{s}_t$ will be small and the effective learning rate will be higher, overcoming the problem in Adagrad.

Next, we look at the Adam Optimizer which combines the ideas of adaptive learning-rate scheduling along with momentum.

15.4 Adam Optimizer

Adam combines ideas from RMSProp and momentum. The definitions of s_t and v_t are

$$\mathbf{s}_t = \gamma \mathbf{s}_{t-1} + (1-\gamma)\mathbf{g}_t \odot \mathbf{g}_t$$
$$\mathbf{v}_t = \beta \mathbf{v}_{t-1} + (1-\beta)\mathbf{g}_t$$

where $\beta, \gamma \in [0, 1]$
Redefine the weight update as

$$\mathbf{w}_t \leftarrow \mathbf{w}_{t-1} - \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \odot \mathbf{v}_t \tag{6}$$

As you can see, the $\mathbf{v}_t$ part in the update rule is inspired from momentum and the effective learning-rate part is inspired from RMSProp. In the research paper which introduces Adam, there is one more layer of detail. Define $\hat{\mathbf{s}}_t$ and $\hat{\mathbf{v}}_t$ as

$$\hat{\mathbf{s}}_t = \frac{\mathbf{s}_t}{1-\gamma^t}$$
$$\hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1-\beta^t}$$

The weight update will now be

$$\mathbf{w}_t \leftarrow \mathbf{w}_{t-1} - \frac{\eta}{\sqrt{\hat{\mathbf{s}}_t + \epsilon}} \odot \hat{\mathbf{v}}_t \tag{7}$$

Recap

Optimizers

Optimizers are used for providing adapting learning rates during training process. Different Optimizers used are:

- **SGD with momentum**: Concept of Momentum, smoothed past gradients over time
- **AdaGrad**: Adaptive learning rates per feature
- **RMSProp**: Improvement on AdaGrad
- **Adam**: Combination of SGD with Momentum and AdaGrad

16 Bias Correction in Adam

Recall that

$$s_t \leftarrow \gamma s_{t-1} + (1-\gamma) g_t \odot g_t$$
$$v_t \leftarrow \beta v_{t-1} + (1-\beta) g_t$$

Bias Correction for Adam optimizer is as follows:

$$\hat{s}_t \leftarrow \frac{s_t}{1-\gamma^t}, \quad \hat{v}_t \leftarrow \frac{v_t}{1-\beta^t}$$

With bias correction included, the weight update becomes:

$$w_t \leftarrow w_{t-1} - \frac{\eta}{\sqrt{\hat{s}_t} + \epsilon} \odot \hat{v}_t$$

Why do we need bias correction? Let us expand s_t:

$$\begin{aligned}
s_t =& (1-\gamma) g_t \odot g_t + \gamma s_{t-1} \\
=& (1-\gamma) g_t \odot g_t + \gamma(1-\gamma) g_{t-1} \odot g_{t-1} + \gamma^2 s_{t-2} \\
& \vdots \\
=& (1-\gamma) g_t \odot g_t + (1-\gamma)\gamma g_{t-1} \odot g_{t-1} + \cdots + (1-\gamma)\gamma^t s_0 \\
=& (1-\gamma) g_t \odot g_t + (1-\gamma)\gamma g_{t-1} \odot g_{t-1} + \cdots + (1-\gamma)\gamma^{t-1} s_1 \quad [\because s_0 = 0]
\end{aligned}$$

Consider the sum of the coefficients in the above equation:

$$(1-\gamma)(1 + \gamma + \gamma^2 + \ldots + \gamma^{t-1})$$

Firstly, note that for large values of t, we have:

$$1 + \gamma + \gamma^2 + \cdots + \gamma^{t-1} \approx \frac{1}{1-\gamma}$$

Thus, large values of t, the sum of the coefficients is approximately equal to 1.
However, for small values of t, we only have:

$$1 + \gamma + \gamma^2 + \cdots + \gamma^{t-1} = \frac{1-\gamma^t}{1-\gamma}$$

Thus, for small values of t, the sum of the coefficients only equal to $1-\gamma^t$.
To remove this bias, we use a bias correction term for s_t, which is $\hat{s}_t = s_t/(1-\gamma^t)$.

17 Convolutional Neural Networks

Convolutional neural networks are the neural networks which have convolution to be a basic operation on which they operate on (other than activation functions). So what difference does this make ?

Consider the problem of recognizing an object in the image, we would consider this image to be in the form of stacked pixels. In this case, we would like our model to identify or use locality that is the model should be able to learn about a pixel from its neighbouring pixels. The model should be invariant to translation, that is the position of a object in the image shouldn't matter to the model. Also, the model should be parameter efficient.

Fully connected(dense) layers have no awareness of spatial information. The key concept behind the convolutional layers is to use kernels or filters to detect spatial information. Kernels can be understood as small arrays (typically squares) of learnable weights or parameters. These filters slide across an input to detect spatial patterns in local regions. Also, the only learnable weights in this case are the weights in filter. Observing locality and translation invariance are two important features of CNNs which makes it a good choice for pattern recognition or computer vision problems.

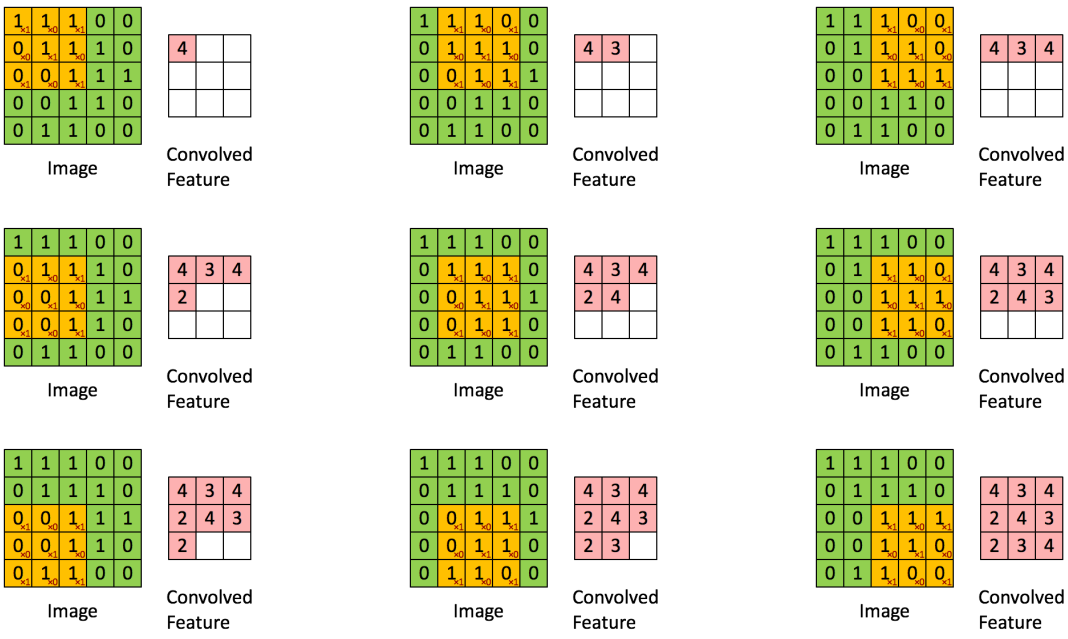
Desired Features	Feedforward neural networks	Convolutional Neural Network
Locality	✗	✓
Translation Invariance	✗	✓
Parameter Efficient	✗	✓

Table 1: Desired Features: FFN vs CNN

18 Convolution Operation

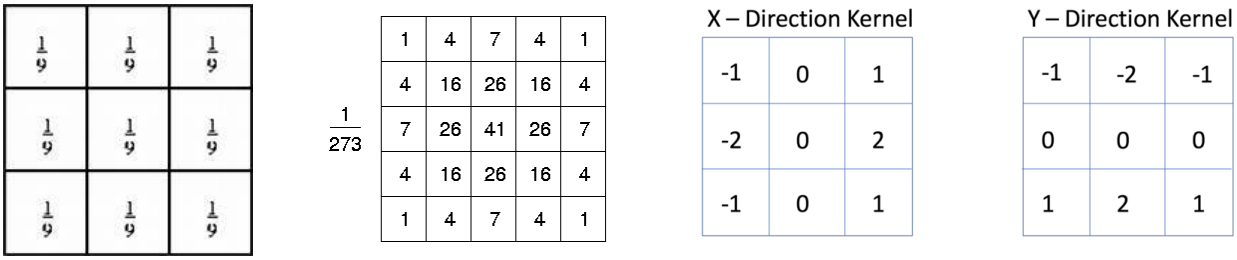
In a convolution operation, a kernel (also called a filter) is moved along the image to get convolved features. When convolving, you compute the dot product between all elements of the kernel with the value of the image underneath the kernel (in essence, taking a weighted sum over the region). This is the main operation underlying CNNs and is called convolution. (Technical Aside: This is not technically convolution, it's instead called cross-correlation.)

Here is an example below:



These filters can capture different features of the image.

- **Mean Kernels**: It averages all underlying elements of input image resulting in a blurred image. This helps retain important information and removes contrast information in image leading to a smoothed image.
- **Gaussian Kernels**: It gives more weight to pixels closer to the center of the kernel and less weight to pixels farther away. This results in a reduction of high-frequency noise and fine details in the image. It also gives a blurred image but is more effective at preserving fine details compared to mean kernel.
- **Edge Detection kernels**: It identifies abrupt changes in pixel intensities, highlighting the object boundaries and transitions in an image. Also called *Sobel filters*.



((a)) Mean Kernels

((b)) Gaussian Kernels

((c)) Edge Detection Kernels

Figure 13: Different Types of Filters

When we design filters for convolutional layers, we will have to decide filter size, stride and padding for the filter. Filter size is the size of the filter (since filters are square, the size of the filter would be the size of the square). Stride is the number of columns/rows to be left or the number of the rows/columns the filter has to be moved when the filter slides through the entire image. Stride would determine how fast the convolution is done. Padding tells how much extra pixel columns or rows (of value 0) are to be added to the image in order to change the size of the convolved image. Padding is typically used to preserve the spatial dimensions or to control the size reduction that occurs during convolution.

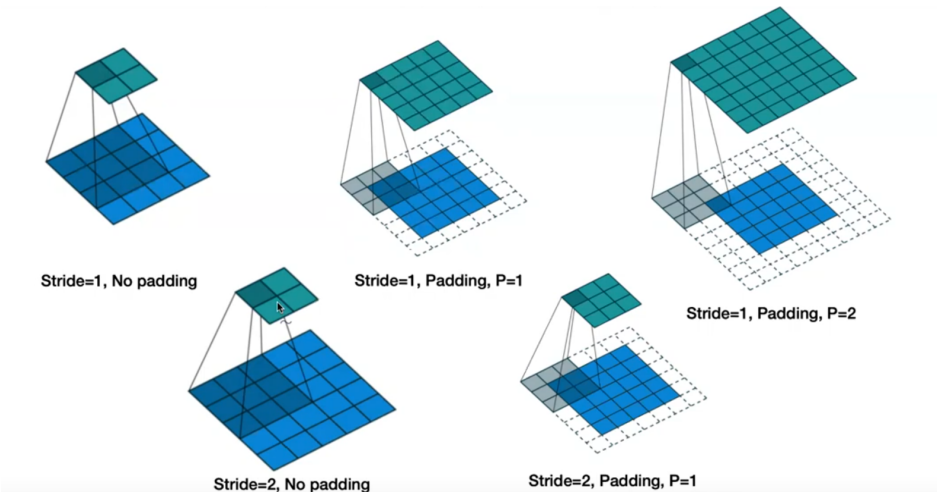


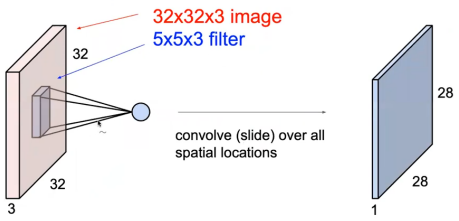
Figure 14: Stride & Padding

19 Components of CNN

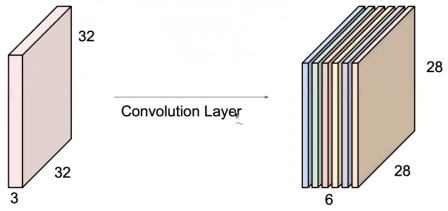
19.1 Convolution Layer

In CNNs, we learn the kernel weights/matrices from data. In a convolution layer, we use multiple kernels. We fix the dimensions of each of these kernels and stack the output from these multiple kernels. Using convolution we could decrease the size of output feature map.

In the figure below, there are 3 channels/volumes in the image and we are using $5 \times 5 \times 3$ sized filter after convolving we get $28 \times 28 \times 1$ sized featured/activation map. Similarly we can use multiple filters.
In second figure we are using 6 different $5 \times 5 \times 3$ filters which results in $28 \times 28 \times 6$ sized activation map.



((a)) Single filter



((b)) Multiple filters

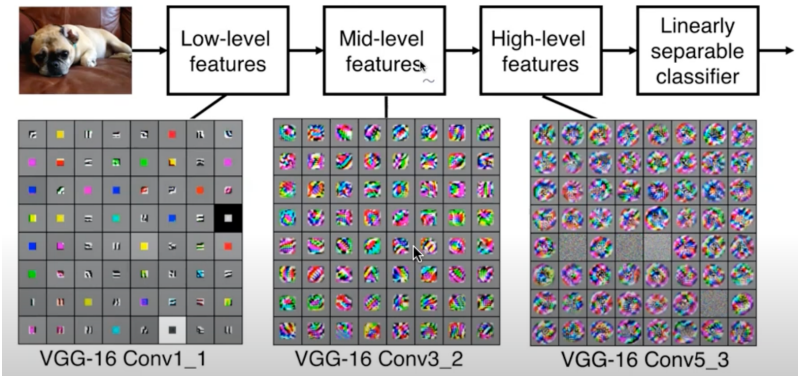
In short, a convolution layer contains multiple (say K) filters of dimension $F \times F$. It may also contain K biases. The number of parameters that have to be learned is $F \times F \times D$ parameters per filter, for K filters and K biases, thus summing up to $F \times F \times D \times K + K$ parameters.

Why do we need different layers?

- With different layers we can have different kernels focusing on different aspects of image which captures different properties that are relevant to final classification task.
- It gives flexibility to learn different properties.

19.2 What do the convolution layers learn

Each convolution layer learns some feature of the input image. The ones at the beginning tries to learn more low-level or basic features, whereas the ones at the end learn more complex or high-level features.



Then these learned high-level features from the last layer are fed into a feed forward network layer for classification.

19.3 Pooling Layer

Pooling layer is used for reducing the dimensions of feature maps. It helps in speeding up the computation and making the model robust. Max Pooling only retains feature with maximum value within the filter. It is one of the most popular pooling methods used in CNNs. Another pooling method used is Average Pooling.

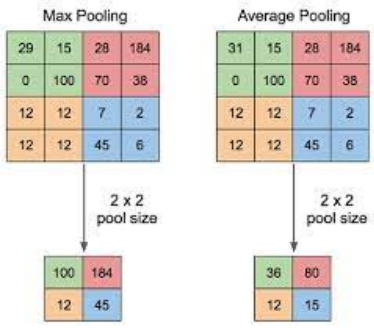


Figure 16: Pooling

NOTE: No learnable parameters are introduced by a pooling layer. Its only role is down-sampling.

20 CNN Size Arithmetic

20.1 Convolution Layers

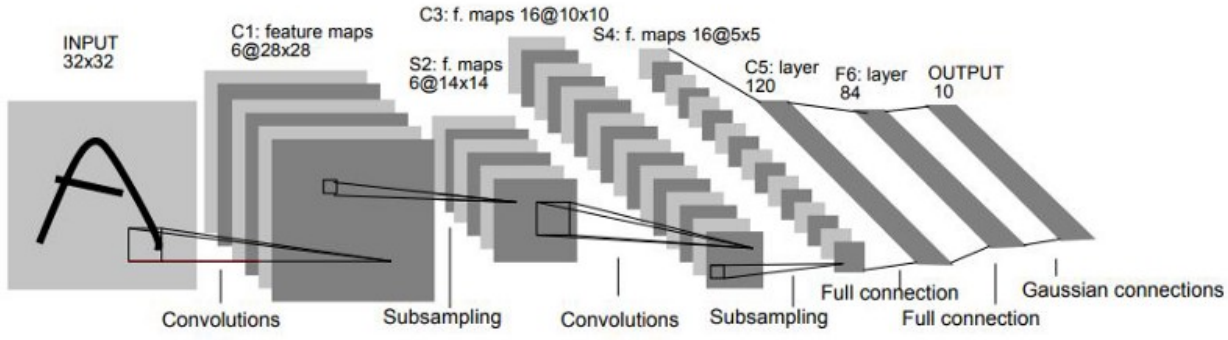
- Accepts a volume of size $W_1 \times H_1 \times D_1$.
- Requires four hyperparameters:
  - Number of filters - K
  - Their spatial extent - F
  - Stride - S
  - Amount of zero padding - P
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = \lfloor (W_1 - F + 2P)/S \rfloor + 1$
  - $H_2 = \lfloor (H_1 - F + 2P)/S \rfloor + 1$
  - $D_2 = K$

  Width and height are computed equally by symmetry.
- With parameter sharing (the same filter used for all regions in the image), we introduce $F \times F \times D_1$ weights per filter, for a total of $(F \times F \times D_1) \times K$ weights and K biases.

20.2 Pooling Layers

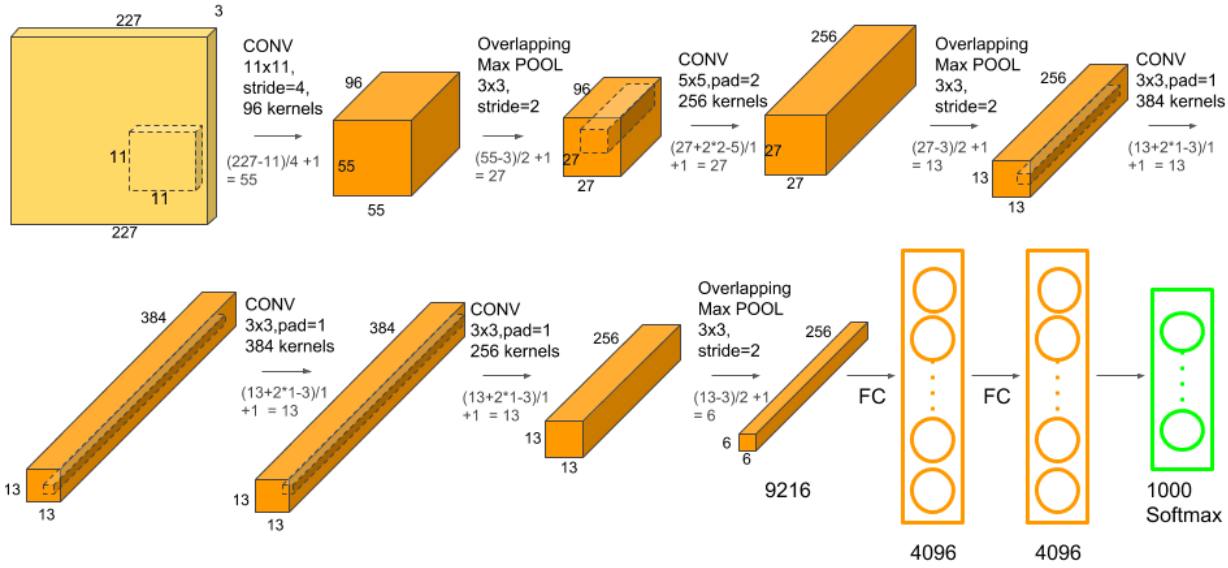
- Accepts a volume of size $W_1 \times H_1 \times D_1$.
- Requires two hyper-parameters:
  - Spatial extent - F
  - Stride - S
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = \lfloor (W_1 - F)/S \rfloor + 1$
  - $H_2 = \lfloor (H_1 - F)/S \rfloor + 1$
  - $D_2 = D_1$

21 LeNet-5 Architecture



- This was one of the first successful CNN architectures.
- It was used to classify handwritten digits.
- Led to the famous MNIST handwritten digits dataset.

22 AlexNet Architecture



- Another successful CNN architecture.
- Had better prediction accuracy than LeNet-5.

23 Batch Normalization

23.1 Motivation

Recall, from linear regression, that preprocessing the inputs to have zero mean and unit variance (normalization) was the typical first step in the pipeline. *Why?*

- It helps constrain the function complexity of the hypotheses in the hypothesis plane.
- Standardization works well with gradient descent style optimizers by putting all parameters apriori in a similar space.

Internal Covariate Shift

In deep neural networks, variables in intermediate layers take values of varying magnitudes. When the parameters (variables) of a layer change, so do the distributions of inputs to the subsequent layers.

Internal covariate shift is defined as the change in the distribution of network activations due to the change in network parameters during training. One approach to address this issue is to use adaptive solvers like Adagrad, while an alternative method involves employing adaptive normalization techniques such as batch normalization.

Sergey Ioffe and Christian Szegedy proposed **Batch Normalization** to mitigate the problem of internal covariate shift, which can cause convergence challenges in deep neural networks, by re-centering and re-scaling the inputs of each layer.

Note: It is argued that batch normalization does not reduce internal covariate shift, but rather makes the overall training dynamics smoother, which improves the performance.

23.2 Procedure

Batch normalization has two steps:

1. In each training iteration, from each layer's activations, subtract the mean and divide by the standard deviation.
2. After normalizing, apply a scaling coefficient and shift the inputs by an offset (both are learnable parameters).

The mean and standard deviation for a batch are computed using mini-batch statistics during training.
Let $\mathcal{B}$ denote a batch and let $\mathbf{x} \in \mathcal{B}$. $\text{BN}(\mathbf{x})$ can be written as

$$\boxed{\text{BN}(\mathbf{x}) = \gamma \odot \frac{\mathbf{x} - \hat{\mu_{\mathcal{B}}}}{\hat{\sigma_{\mathcal{B}}}} + \beta}$$

Batch mean:

$$\hat{\mu_{\mathcal{B}}} = \frac{\sum_{\mathbf{x} \in \mathcal{B}} \mathbf{x}}{|\mathcal{B}|}$$

Batch variance:

$$\hat{\sigma_{\mathcal{B}}}^2 = \frac{\sum_{\mathbf{x} \in \mathcal{B}} (\mathbf{x} - \hat{\mu_{\mathcal{B}}})^2}{|\mathcal{B}|} + \epsilon$$

Here, γ is the scaling coefficient, and β is the offset. These parameters are learned in the optimization process. ϵ (an arbitrarily small constant) is added for numerical stability.

At test time, we use the running averages of the mean and standard deviation values obtained during training.

Batch normalization is typically applied after the affine layer (fully connected layer) but before the non-linear activation.

23.3 Benefits of Batch Normalization

- Improves gradient flow through the network, makes more activation functions viable.

- Allows us to use much higher learning rates and be less careful about initialization.
- It also acts as a regularizer, in some cases eliminating the need for Dropout.

Batch normalization makes overall training dynamics smoother, doesn't necessarily reduce internal covariate shift. Batch normalization have worked best for CONV and regular deep Neural Networks. Other popular regularization techniques include the use of **Layer Normalization, Group Normalization**; **Residual Connections** help enable stable learning in very deep architectures.

23.4 Other Normalizations

- **Layer normalization**: Layer normalization normalizes the activations of a layer across all units in the layer. Layer normalization doesn't depend on batches and the normalization is per data point, so the exact same calculations are done during training time and test time.

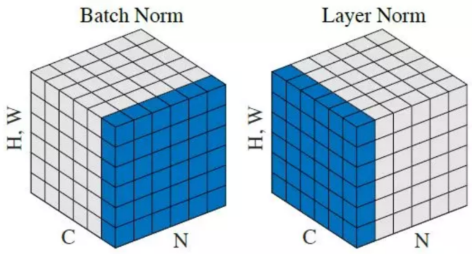


Figure 17: Layer Normalization

- **Group normalization**: Group normalization is a more generalized version of layer normalization, it divides the features into groups and normalizes the features within each group independently. The number of groups is a hyperparameter specified prior to training.

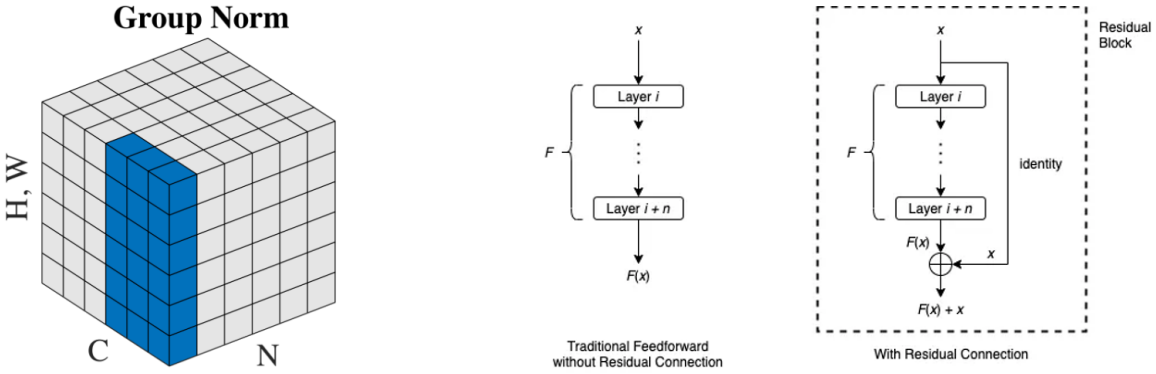


Figure 19: Residual Block

- **Residual connections**: Residual connections provide another path for data to reach the latter parts of the neural network by skipping some layers. It addresses the vanishing gradient problem and has led to the development of extremely deep networks, such as ResNets, that have achieved state-of-the-art results in many tasks.

24 Recurrent Neural Networks

So far, in feed-forward neural networks and CNNs, we have dealt with fixed-size inputs and fixed-size outputs. However, we often need to deal with inputs and outputs of varying lengths. RNNs come to our rescue here.

RNNs are neural networks that deal with sequential data and also give a variable-sized output. RNNs are distinguished by their memory, as they (unlike classical NNs and CNNs) take information from prior inputs

and outputs to influence the current output. This memory is maintained through a hidden state.

24.1 Language Modelling

The language modeling problem focuses on predicting the next word in a sentence; given a word history $w_1, w_2, \ldots, w_{t-1}$, we want to find w^*

$$w^* = \arg\max_w P(w|w_1, w_2, \ldots, w_{t-1})$$

.

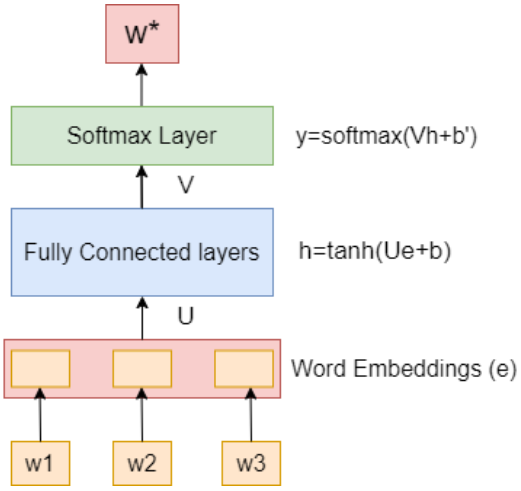


Figure 20: Language modeling using feed-forward neural networks

We cannot directly work with words so we use word embeddings. An embedding of a word is a representation of the word in a d-dimensional vector space. If we were to use basic feed-forward neural networks for the language modeling problem, we could only look at some n length history because feed-forward neural networks have fixed size inputs, so we lose information about the previous words. Such a model makes an n^{th} order Markovian assumption, that given the previous $n-1$ words, the probability of the n^{th} word is independent of the words prior to those $n-1$ words. So after predicting one word, we slide our window of inputs to predict the next.

However, RNNs that can work with variable-length inputs are more suited for this task.

24.2 RNN Architecture

RNNs maintain a hidden state, which remembers the information about past inputs. The output is predicted using the hidden state for the current time stamp. For each input, it uses the same parameters to predict the output. This reduces the complexity of the model, unlike other neural networks.

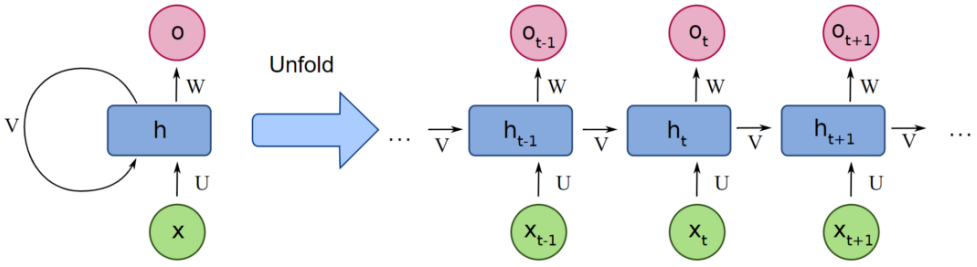


Figure 21: RNN Architecture

Consider a sample RNN for a classification task

$$\mathbf{h}_t = \tanh(\mathbf{U}\mathbf{x}_t + \mathbf{V}\mathbf{h}_{t-1} + \mathbf{b})$$

$$\mathbf{o}_t = \mathbf{W}\mathbf{h}_t + b'$$

$$\hat{\mathbf{y}_t} = softmax(\mathbf{o}_t)$$

Here $\mathbf{U}, \mathbf{V}, \mathbf{W}, \mathbf{b}, b'$ are the parameters of the model which are shared across time steps.
$\mathbf{h}_t$ is the hidden state at the t^{th} timestep
$\mathbf{x}_t, \mathbf{o}_t$ are the input and output at the t^{th} timestep
$\mathbf{y}_t$ is the prediction probability distribution at the t^{th} timestep

24.3 Vanishing/Exploding Gradients

RNNs by design model temporal dependency, like a very deep neural network. Because of the large depth, the gradient, which is multiplicative across layers can vary greatly in magnitude causing the vanishing gradient or the exploding gradient problem.

The **exploding gradient** problem occurs when the gradients of loss function become very large during backpropagation, it shows that the model is unstable and unable to learn.

The **vanishing gradient** problem occurs when the gradients of loss function become very small as they are backpropagated, it hinders the model's capabilities to learn long-term dependencies.

24.4 Gradient Clipping

Gradient clipping is a technique used to deal with the exploding gradient problem. When performing back propagation we cap the maximum values(norm) for the gradient.

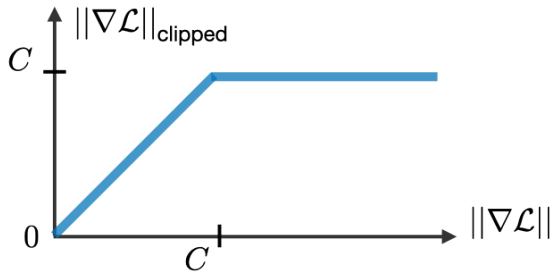


Figure 22: Gradient Clipping

25 Long-Short Term Memory Networks

Long-Short Term Memory Networks (**LSTM**s) is a variant of RNN, used in deep architectures specifically used to address the Vanishing-gradient problem. Unlike RNN, rather than applying an element wise non linearity to the affine transformation of inputs and recurrent units, LSTM consists of **gates** that have an internal recurrence (a self-loop), in addition to the outer recurrence of the RNN.These gates enable LSTM's to both **accumulate** and **forget** states conditioned on the context.

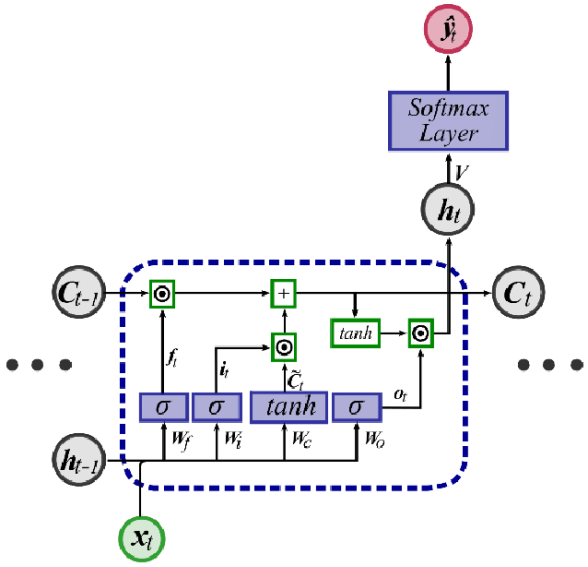
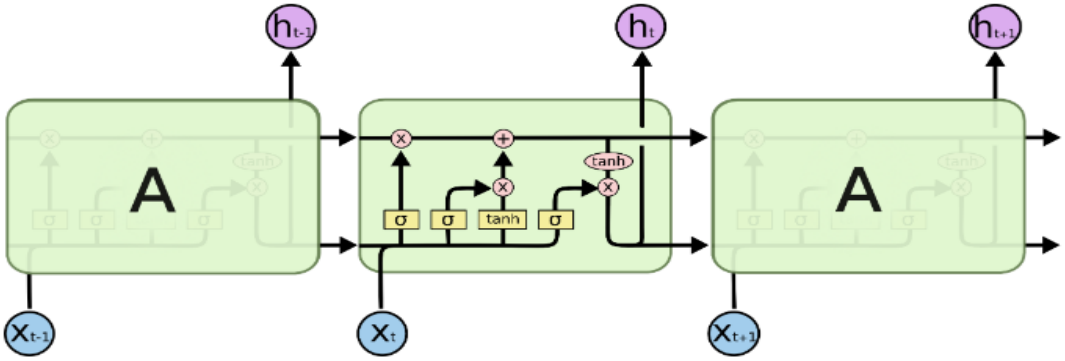


Figure 23: LSTM cell unit

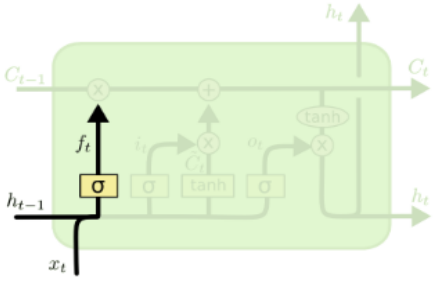


There are two states in LSTM, cell-state C_t and hidden-state h_t. Cell state is a memory of the LSTM cell and hidden state (cell output) is an output of this cell. The LSTM introduces three types of gates—**input gate, output gate, and forget gate**, values come out between 0 and 1. The closer to 0 means to forget, and the closer to 1 means to keep.

25.1 LSTM implementation

25.1.1 Forget gate

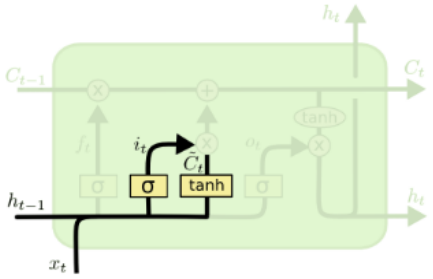
This gate decides what information should be thrown away or kept. The information from the previous hidden state and information from the current input is passed through the sigmoid function. It looks at h_{t-1} and current input x_t and outputs a number between 0 and 1 for each number in the cell state C_{t-1}, indicating whether to retain 1 or discard 0 the corresponding information from the cell state.



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

25.1.2 Input gate

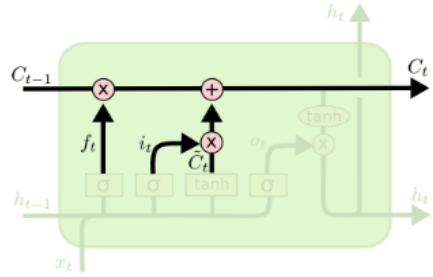
The input gate decides what relevant information can be added from the current step. Sigmoid layer decides which values we'll update. Next, a tanh layer creates a vector of new candidate values, $\tilde{C}_t$, that could be added to the state. In the next step, we'll combine these two to create an update to the state.



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

25.1.3 Cell State

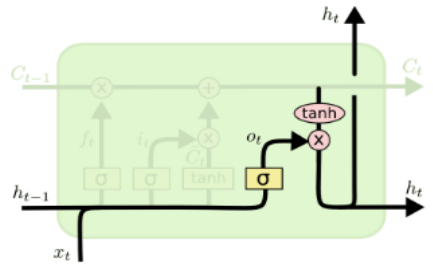
It's now time to update the old cell state, C_{t-1}, into the new cell state C_t. We multiply the old state by f_t, forgetting the things we decided to forget earlier. Then we add $i_t * \tilde{C}_t$. This is the new candidate values, scaled by how much we decided to update each state value.



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

25.1.4 Output gate

The output gate determines what the next hidden state should be using the output and the updated cell-state. Sigmoid layer decides what parts of the cell state we're going to output. Then, we put the cell state through tanh and multiply it by the output of the sigmoid gate.



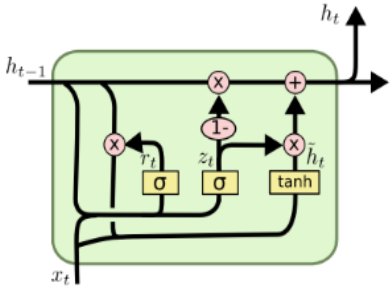
$$o_t = \sigma\left(W_o \; [h_{t-1}, x_t] \; + \; b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

where operator * denotes the Hadamard product, f_t is the forget gate, i_t is the input gate, o_t is the output gate, and C_t and h_t are the updated cell states. The non-linearities used are sigmoid and tanh.

26 Gated Recurrent Unit

A slight variation of the LSTM is the **Gated Recurrent Unit**(GRU), it combines the forget and input gates into a single "update gate." It also merges the cell state and hidden state, and makes some other changes.

The resulting model is simpler than standard LSTM models.



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$