# APS360H1

Applied Fundamentals of Deep Learning

G. Jeon

Last updated: August 20, 2024

## Contents

# ※ 1. Course Introduction

## 1.1 Content

This chapter introduces AI as a concept and goes over the definitions of deep learning, its successes and caveats, its place in society and the basics of machine learning.

## 1.2 What is AI?

What's AI? AI is an attempt to replicate human learning and intelligence. Historically, there have been two approaches, the symbolic and connectionist approach. The symbolic approach defines AI as being program driven and that it constructs discrete changes and outputs (Data + Results = Program). The connectionist approach defines AI as being observation driven and that it constructs outputs (Data + Results = Program).

## 1.3 What is Machine Learning?

Machine learning is a broad term, but essentially we are defining it as a program that will take in datasets to output a specific solution.

## 1.4 Why care?

We want to solve problems. Some problems are more complex than others. These problems have sets of rules attached to them and we want to make solutions that we can keep learning from.

## 1.5 Formal Definition of ML

"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E." - Mitchell et al. 1997

## 1.6 Formal definition of DL

"Deep learning is a subset of machine learning that allows multiple levels of representation, obtained by composing simple but non-linear modules that each transform the

representation at one level (starting with the raw input) into a representation at a higher, slightly more abstract level." - LeCun et al. 2015

## 1.7 Some successes and caveats in DL

Some success and caveats of DL in industry/application:

Successes:

- Image generation
- Math applications
- Coding solutions
- Simulators

Caveats

- Interpretability
- Causality
- Fairness & Bias

## 1.8 Machine learning basics

We work with three types of learning methods:

**Supervised learning**: Regression or classification from outputs

**Unsupervised learning**: Little to no oversight, requires observation from human annotations

**Reinforcement learning**: Rewarded from certain actions

We can probably assume we need good assumptions. The inductive bias is the set of assumptions that's used for modeling and learning.

Take the fitting example above. We can use the mean squared error to find the best line of fit, minimizing error. We want to make a program that can replicate what we do to select the best line of fit. But sometimes we can run into errors and discrepancies. We can run into generalization errors where the model overfit or underfit the best line of fit. We want the model to constantly play this game of balance to learn from the data but also prevent such errors from occuring in the future.

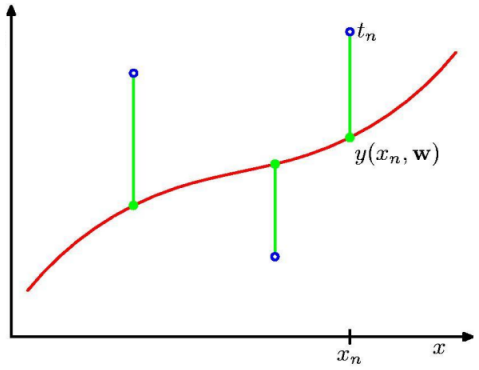



*Figure 1: MSE example*

## ※ 2. Artificial Neural Networks

### 2.1 Content

This chapter introduces Artificial Neural Networks (ANNs), the fundamental building blocks for understanding the course. It covers neurons, activation functions, training, loss functions, gradient descent, and network architectures.

### 2.2 The Neuron

Neurons, the basis of neural networks, mimic biological structures. A neuron comprises dendrites (receivers), cell bodies (integrators), axons (transmitters), and synapses (connections). Together, they process stimuli, leading to reactions like smiling at a waving friend or crying from pain, enhancing our understanding of the world.

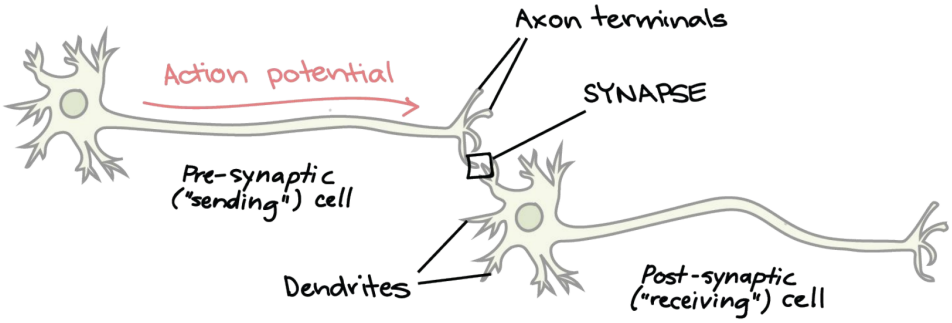



*Figure 2: Mechanics of a neuron*

### 2.3 Artificial Neuron

Artificial neurons mimic biological ones by assigning variables to emulate their components, encapsulated into the following familiar equation.

$$y = f(\mathbf{w} \cdot \mathbf{x} + b) \tag{1}$$

where

| | |
|---|---|
| $\mathbf{x_i}$ | is the input vector (user input, text file, unicode, etc) |
| $\mathbf{w_i}$ | is the weight vector for the input $\mathbf{x_i}$ |
| $b$ | is the bias, a weight with no input |
| $f$ | is the activation function that determines the output |
| $y$ | is the output (such as the class an image belongs to) |

These variables fall into place as a summation function (hence subscript $i$).

### 2.4 Activation Functions

Suppose the activation function is a simple linear function.

$$y = f(\mathbf{w} \cdot \mathbf{x} + b) \longrightarrow y = \mathbf{w} \cdot \mathbf{x} + b$$

This is a special equation, and it's more clear if we write it out for a 2-D plane.

$$y = w_0 x_0 + w_1 x_1 + b \tag{2}$$

Here, $y = \mathbf{w} \cdot \mathbf{x} + b$ is a generalized line for any dimension, known as a hyperplane, splitting the n-dimensional input space into 2. This means that data will plot itself above or below the equation.

### 2.5 Neuron Activation Function

Real datasets usually can't be separated by a straight line on a graph. So, we use non-linear transformations. This involves adding layers that twist and reshape the data, instead of just stacking straight lines on top of each other.

### 2.6 Early Activation Functions: Perceptrons

Early artificial neurons used basic binary activation functions like sign and Heaviside functions. The problem with these functions is that they're piecewise, which means they're not differentiable, continuous, or smooth. These are known as decision boundaries.

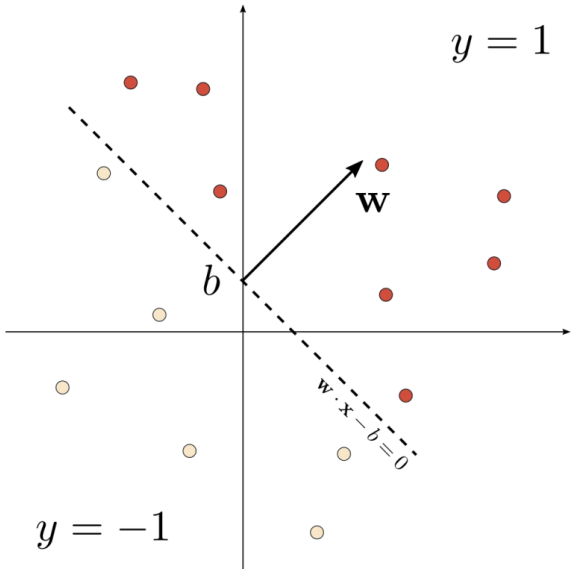



*Figure 3: Our data plots itself above and below our line equation*

$$f(x) = \text{sign}(x) \qquad \text{Sign function} \tag{3}$$

$$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases} \qquad \text{Heaviside (unit) step function} \tag{4}$$

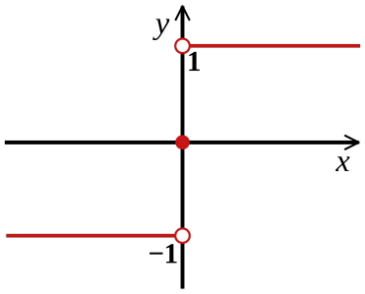



*Figure 4: Heaviside function*

Sigmoid activation functions, such as hyperbolic tangents and logistic functions, replaced early activation functions. They are smooth, continuous, and easily differentiable, with a range typically between [-1, 1] or [0, 1]. Though other options exist, sigmoids remain effective in practice.

$$f(x) = \tanh(x) \qquad \text{Hyperbolic tangent} \tag{5}$$

$$f(x) = \frac{1}{1 + e^{-x}} \qquad \text{Logistic function} \tag{6}$$

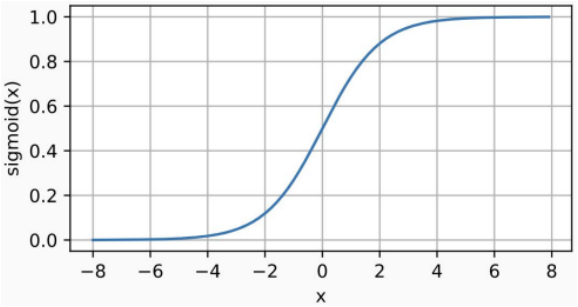



*Figure 5: Sigmoid function*

Sigmoid activation functions can produce gradients, but saturated neurons may cause them to vanish very quickly away from $x = 0$. This is why modern deep learning often employs Rectified Linear Unit (ReLU) based activation functions.

$$\text{ReLU}(x) = (x)^+ = \max(0, x) \qquad \text{ReLU} \qquad (7)$$

$$\text{Leaky ReLU}(x) = \begin{cases} x & \text{if } x \geq 0 \\ \text{negative\_slope} \times x & \text{otherwise} \end{cases} \qquad \text{Leaky ReLU} \qquad (8)$$

$$\text{PReLU}(x) = \begin{cases} x & \text{if } x \geq 0 \\ ax & \text{otherwise} \end{cases} \qquad \text{Parametric ReLU} \qquad (9)$$

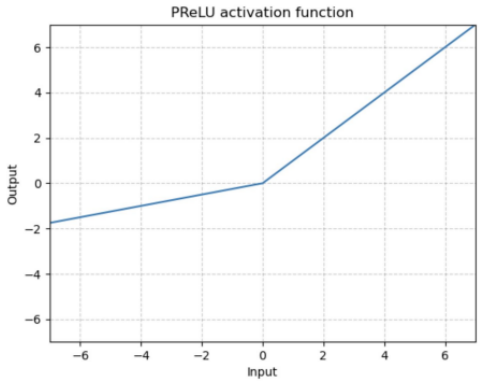



*Figure 6: Parametric function*

We can approximate ReLU activation by continuous functions. These perform on par or better than ReLU functions.

$$\text{SiLU}(x) = x.\sigma(x) = \frac{x}{1 + e^{-x}} \qquad \text{SiLU (Swish)} \qquad (10)$$

$$\text{SoftPlus}(x) = \frac{1}{\beta} \log(1 + e^{\beta x}) \qquad \text{Soft Plus} \qquad (11)$$

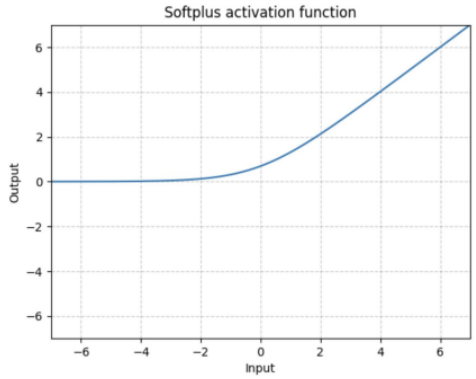



*Figure 7: Soft Plus function*

## 2.7 Training Neural Networks

How do we learn the weights and bias of a neural network, given that our model takes the form of $f(wx + b)$? We use our error to decide how to change the weights via an iterative process till an acceptable error is reached.

1. Make a prediction for some input x, with a known correct output t
2. Compare the correct output with predicted output to compute loss
3. Adjust the weights and/or bias to minimize the gap between correct output with predicted output
4. Repeat until the acceptable level of error is reached

Now, let's consider performing a 'forward pass' on the network. The 'forward pass' refers to the calculation process that computes the values of the output layers from the input data, traversing through all the neurons. The 'backward pass,' on the other hand, refers to the process of adjusting the weights (effectively learning) by using the gradient descent algorithm (or similar methods) to minimize errors.

## 2.8 Loss Functions

A loss function computes how predictions compare to the ground truth labels. A large loss differs from the ground truth whereas the small loss matches the ground truth. We're specifically after the error over all training samples, i.e. average error.

Let's say we pass some input through an activation function and it outputs a set of arbitrary values, or *logits*. How do we decide which class of number to use? If we wanted to use probabilities, we could use the softmax function.

### 2.8.1 Softmax Function

$$\text{Softmax}(x)_i = \frac{e^x}{\sum_{k=1}^{K} e^{xk}} \tag{12}$$

Where the inputs, $x$, are the logits, and the outputs are the probabilities, $p(Class)$.

### 2.8.2 One-hot Encoding

Another way to do it would be to use the method of one-hot encoding, taking the form of a unique vector for each different class. This method just maps the classes to vector representations

$$\begin{pmatrix} \text{Cat} \\ \text{Dog} \\ \text{Sheep} \end{pmatrix} \xrightarrow{\text{One-hot}} \begin{pmatrix} [1 & 0 & 0] \\ [0 & 1 & 0] \\ [0 & 0 & 1] \end{pmatrix}$$

### 2.8.3 Mean Squared Error

Used with regression problems, we use the mean squared error method.

$$\text{MSE} = \frac{1}{N}\sum_{n=1}^{N}(y_n - t_n)^2 \tag{13}$$

Where $N$ is the number of training samples, $y_n$ is the prediction, and $t_n$ is the respective truth label.

### 2.8.4 Cross Entropy

For classification problems, we use cross entropy.

$$\text{CE} = -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} t_{n,k}\log(y_n, k) \tag{14}$$

Where $N$ is the number of training samples, $K$ is the number of classes, $y_{n,k}$ is the prediction, and $t_{n,k}$ is the respective truth label.

## 2.9 Forward/Backward-Pass and Error Calculations

Let's put our knowledge into practice with a simple Artificial Neural Network (ANN) using a $4 \times 3$ dataset. We have 4 ground truths (training examples) and 3 weights. For simplicity, we'll omit bias terms. Using this simple ANN, we'll model $wx + b$ and compare it with our ground truths.

```
import math

x = [[1.0, 0.1,-0.2],   # data
     [1.0,-0.1, 0.9],
     [1.0, 1.2, 0.1],
     [1.0, 1.1, 1.5]]
t = [0, 0, 0, 1]        # labels
w = [1, -1, 1]          # initial weights

def simple_ANN(x, w, t):
    total_e, e, y = 0, [], []
    for n in range(len(x)):
        v = 0
        for d in range(len(x[0])):
            v += x[n][d] * w[d]
        y.append(1/1+math.e**(-v))     # sigmoid
        e.append((y[n]-t[n])**2)       # MSE
     total_e = sum(e)/len(x)
     return (y, w, total_e)
```

*Figure 8: ANN using MSE*

The `simple_ANN()` function initializes error metrics and model outputs. It iterates over 4 training examples using nested `for` loops to process each dimension. Outputs pass through a sigmoid activation function, and mean-squared error is computed. The function returns the average error by dividing summed errors by the number of training examples.

Here's another implementation of the simple ANN for Forward-Pass & Backward-Pass. Notice the use of iterations, `iter`, and the learning rate, `lr`.

```
def simple_ANN(x, w, t, iter, lr):
    total_e = 0
    for i in range(iter):
        e, y = [], []
        for n in range(len(x)):
            v = 0
            for d in range(len(x[0])):
                v += x[n][d] * w[d]
            y.append(1/1+math.e**(-v))     # sigmoid
            e.append((y[n]-t[n])**2)       # MSE

            # gradient descent to update weights
            for p in range(len(w)):
                d = 2*x[n][p]*(y[n]-t[n])*(1-y[n])*y[n]
                W[p] -= lr*d
     total_e = sum(e)/len(x)
     return (y, w, e)
```

*Figure 9: ANN for forward-backward passes*

## 2.10 Gradient Descent

Let's think of a simple neural network. How can we modify the weights, $w_{ji}$, to reduce error, $E$? First we need to find the how the error is affected by changes in weight. To do this we use the following partial derivative.

$$\frac{\partial E}{\partial w_{ji}} \longrightarrow \frac{\partial L}{\partial w_{ji}}$$

Where $E$ represents the loss, multilayered networks may use the loss, $L$, of the previous layer. But for this case, $E$ is used.

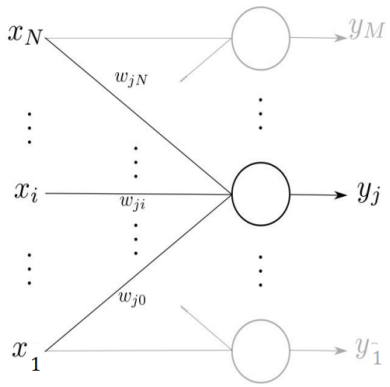



*Figure 10: Simple neural network*

This vector of partial derivatives for all weights is known as the *gradient*, and it determines both the direction of fastest increase in the function and its magnitude, representing the rate of increase. Adjusting the weights according to the slope (gradient) will guide us towards the minimum (or maximum) acceptable error.

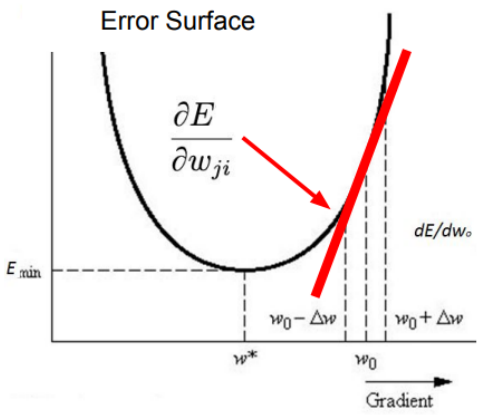



*Figure 11: Error surface graph demonstrating the gradient*

## 2.11 Neural Network Architectures

### 2.11.1 Multiple Layers: XOR

For most problems, having a single NN layer (decision boundary) won't suffice as a solution. This is why we'll need at least one hidden NN layer. A famous problem is the XOR function and how we can't draw a single decision boundary.

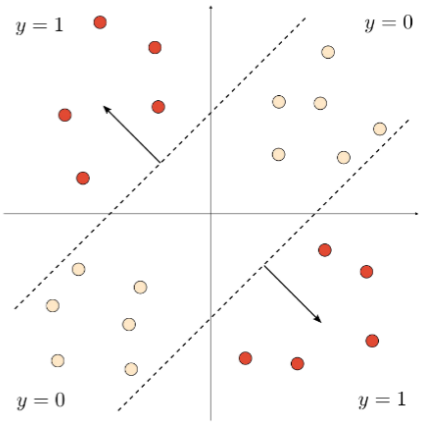



*Figure 12: XOR function*

### 2.11.2 Backpropogation

Backpropagation enhances neural network reliability by tracing errors from outputs to inputs. It adjusts network weights based on previous error rates, iteratively reducing errors to improve reliability.

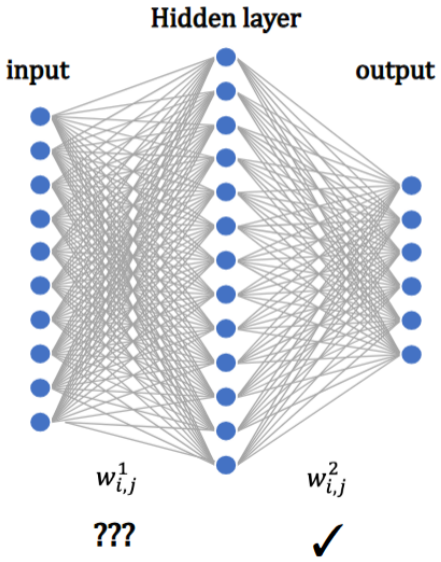



*Figure 13: Back propogation*

# ※ 3. Training Artificial Neural Networks

## 3.1 Content

This chapter covers hyperparameters, optimizers, learning rates, how we normalize our data, and how to regularize our data.

## 3.2 Hyperparameters

Hyperparameters are the architecture choices we have when building a model. Things such as number of layers, the layer size, learning rate, and the different types of activation functions are all examples of hyperparameters.

We have to tune these hyperparameters in order to improve our NN's accuracy and efficiency. Doing will produce optimal results.

## 3.3 Optimizers

Now you may start to notice that the neural network transitions from a learning problem to an opimization problem. With each iteration of change and test, the optimizer determines how each parameter (weight) should change. This course' optimizers will be based on the process of gradient descent. Note, PyTorch automates the gradient computation.

### 3.3.1 Stochastic Gradient Descent (SGD)

With each iteration, computing the gradient takes less time but won't always be faster. SGD allows to conduct a global search to find our optimum, i.e. GD on the entire training data.

### 3.3.2 Mini-Batch Gradient Descent

We can also conduct GD on batches. Such process is as follows:

1. Use our network to make the predictions for n samples
2. Compute the average loss for those n samples
3. Take a "step" to optimize the average loss of those n samples

### 3.3.3 Ineffective Batch Size

What if the batch size is too small/big? If it's too small we optimize a different loss function at each iteration, this can cause noise. Or if it's too large the average may not

change too much and not worth iterating through.

### 3.3.4 Gradient Descent: N-Dimensional and SGD with Momentum

Earlier we discussed the topic of deep neural networks. These are networks that can consist anywhere from millions to billions of parameters. Real gradient descent of a deep network is opimization across and in millions of dimensions.

Recalling our learning in ODEs, you may notice that the existence of zero points are saddle points. These plateaus are a problem but can be addressed using specialized variants on gradient descent.

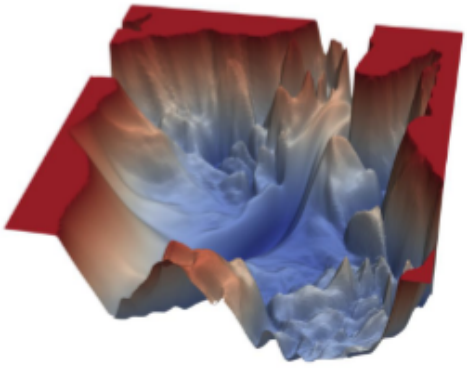


*Figure 14: Each ravine is a zero points, modelled by a saddle point*

As shown on Figure 13, we see a lot of ravines. By definition, they're simply surfaces that curve more steeply in one dimension than in another. SGD has trouble navigating such ravines, causing it to just oscillate across the slopes of the ravine. This is where we employ the use of momentum to help dampen oscillations and point the SGD in the right direction.

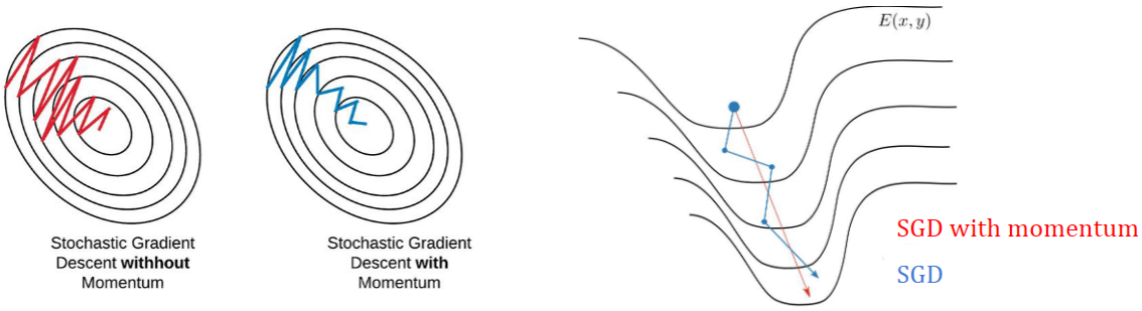



*Figure 15: Simplified mechanical model of when momentum is applied*

This momentum term will increase for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions.

$$\begin{cases} v_{ji}^{t} & = \lambda v_{ji}^{t-1} - \gamma \frac{\partial E}{\partial w_{ji}} \\ w_{ji}^{t+1} & = w_{ji}^{t} + v_{ji}^{t} \end{cases} \tag{15}$$

Same as what we did before except we added a momentum term.

### 3.3.5 Adaptive Moment Estimation (Adam)

Incredibly useful and highly recommended, this optimizer incorporates momentum and an adaptive learning rate, ensuring rapid convergence with minimal tuning.

Adaptive learning rates → each weight has its own learning rate.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \frac{\partial E}{\partial w_{ji}} \tag{16}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) \frac{\partial E}{\partial w_{ji}}^2 \tag{17}$$

$$w_{ji}^{t+1} = w_{ji}^{t} - \frac{\gamma}{\sqrt{v_t} + \epsilon} m_t \tag{18}$$

## 3.4 Learning Rate

The learning rate determines the size of the step that an optimizer takes during each iteration.

$$w_{ji}^{t+1} = w_{ji}^{t} - \gamma \frac{\partial E}{\partial w_{ji}} \tag{19}$$

We approach tuning our learning rates like Goldilocks and the Three Bears.

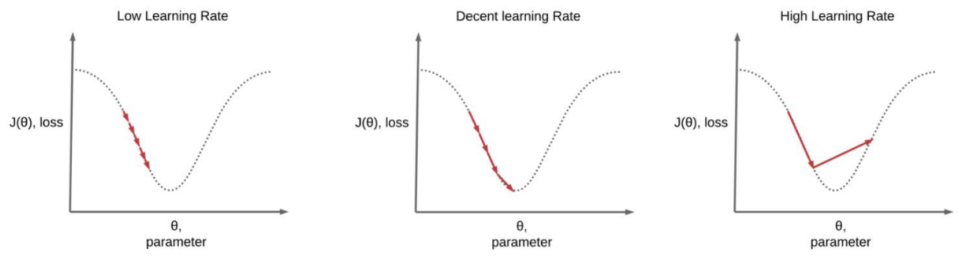



*Figure 16: Not too high, not too low, just right*

## 3.5 Normalization

When dealing with large datasets, it helps to remove the impact of scale and put all features on the same scale. This is called normalization.

$$X_i = \frac{X_i - \mu_i}{\sigma_i} \tag{20}$$

This only normalizes the first layer because we treat the x variable as the input. To normalize subsequent layers we use batch normalization. This just takes the output of a previous activation layer by subtracting the batch mean and dividing by the batch standard deviation.

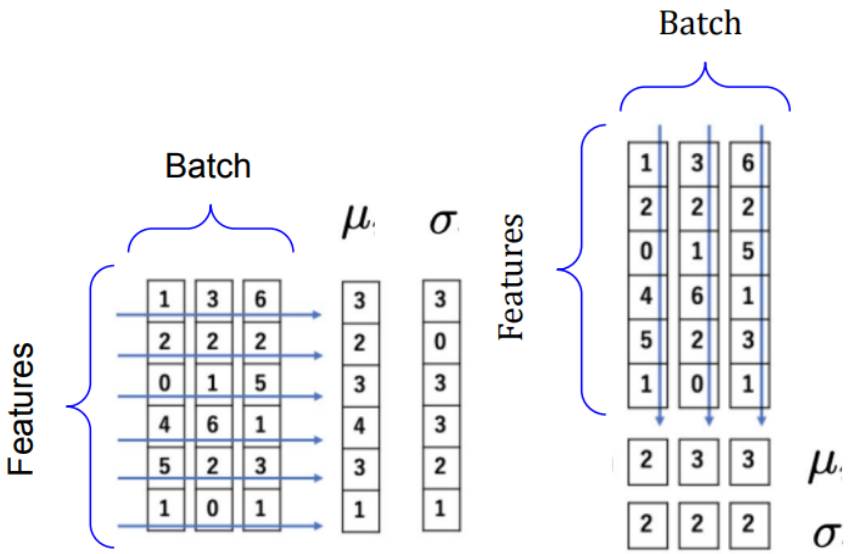



*Figure 17: Batch and layer normalization*

As you can guess, using a higher learning rate can speed up training, regularize the model, and make it less sensitive to initialization; however, its effectiveness depends on batch size, with no impact on small batches, and it is incompatible with stochastic gradient descent (SGD).

Another process we can use is layer normalization. Unlike batch normalization, layer normalization is simple to implement with no moving parameters or averages and isn't dependent on batch size.

## 3.6 Regularization

### 3.6.1 Dropout

During randomized training, we will have to disregard certain nodes in a layer at random during training in order to ensure that no nodes are codependent with one another. This is called dropout.

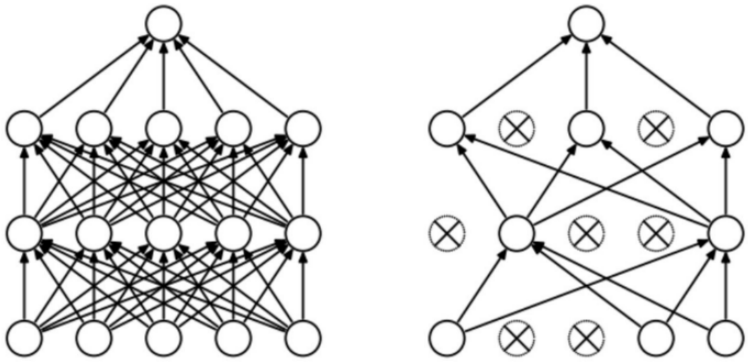



*Figure 18: Dropout regularization*

## ※ 4. Convolutional Neural Network I

### 4.1 Content

This section covers the convolutional operator, CNNs, the motivation behind using CNNs, pooling operators, and the PyTorch implementation.

### 4.2 Motivation

Previously we established the usefulness of ANNs however we'll reach a point where we'll start getting some issues. If we make our NNs bigger, the computational complexity will increase, will use bad inductive bias, and won't be flexible to work with. To avoid such hurdles we'll have to implement CNNs.

### 4.3 Convolution Operator

A convolution is a mathematical operation on two functions $f$ and $g$ that expresses how the shape of one is modified by the other. A convolution in one-dimension is as follows.

$$(f * g)[n] = \sum_{k=-\infty}^{\infty} f[k]g[n-k] \tag{21}$$

But for applications such as images, we need the add another dimension, i.e. a convolution of Image **I** with filter Kernel **K**.

$$y[m,n] = I[m,n] * K[m,n] = \sum_{j=-\infty}^{\infty} \sum_{i=-\infty}^{\infty} I[i,j].K[m-i,n-j] \tag{22}$$

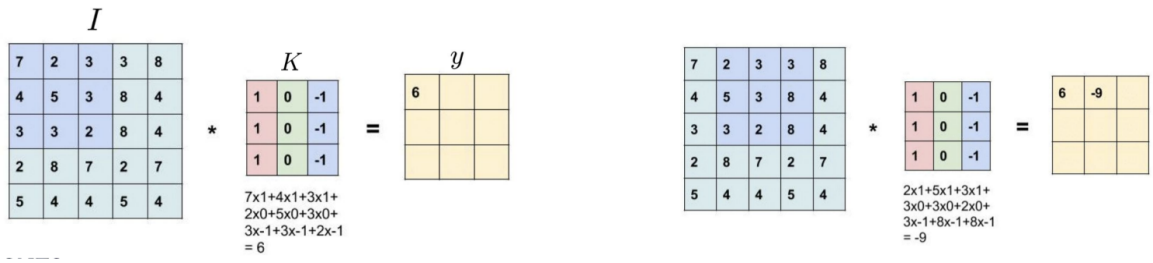



*Figure 19: Visual of executing Formula 22*

The need to apply filters and modify large pools of image data input values are the reason why CNNs are so useful. We went from having to implement these kernels by hand to developing and using classical computer vision via multi-stage feature (kernel) engineering.

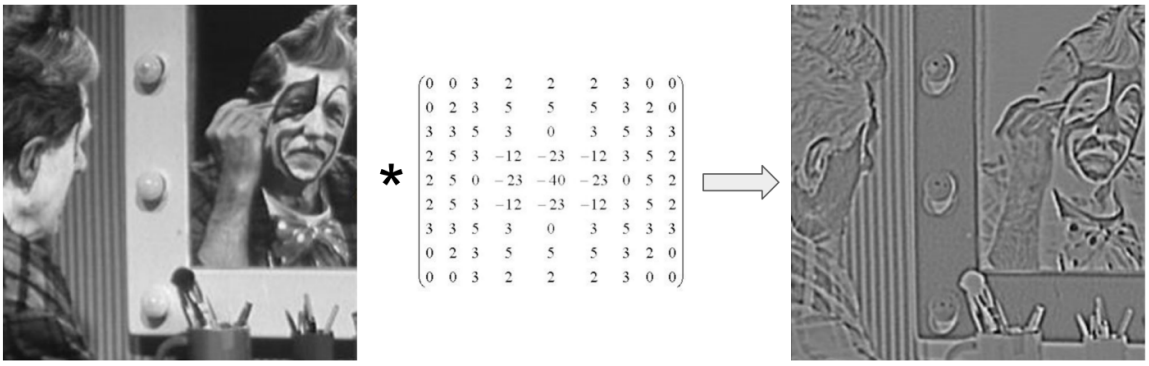


*Figure 20: Blob detector*

Figre 20 shows an implementation of a blob detector. Other such implements include blurs, and vertical and horizontal edge detectors.

### 4.4 Convolutional Neural Networks

With the following established, we can now understand why we use neural networks: so that we don't have to apply and execute the filters by hand. We can employ our neural network to detect features such as blobs and edges, i.e. from concrete to extremely abstract detection.

#### 4.4.1 Forward & Backward Pass

When training our CNN model we'll initialize the kernels at random. In the forward pass, we get the convolutional filter to pass over the entire image with the kernel at its current state. In the backward pass, taking our resultant output, we update our kernel using the gradient. Then we iterate.

### 4.4.2 Zero Padding

Add zeros around the border of the image before convolution. This is to keep the width and height consistent with the previous layer as work within the bounds of the image's border.

### 4.4.3 Stride

Think of a two-step shift between positions of the kernel.

### 4.4.4 Computing Output Size

For each dimension of an image, the size of the output is computed by the following.

$$o = \frac{i + 2p - k}{s} + 1 \tag{23}$$

Where $i$ is the dimension size, $k$ is kernel size, $p$ is padding size, and $s$ is the size of the stride. A matter of plug and solve.

### 4.4.5 CNN on RGB

When it comes to colour data, images will have an array of RGB values. This requires our kernel to become a three-dimensional tensor.

$$\text{dim kernel} = 3 \times k \times k$$
$$\text{dim image} = 3 \times i \times i$$

## 4.5 Pooling Operators

Have you ever wondered why we reduce the number of units before the final output layer? It's to consolidate and narrow down the amount of information. We consolidate this information with convolutional layers via strided convolutions, max pooling, and average pooling.

### 4.5.1 Max Pooling

Pooling layers produce the relevent max pool with the largest values in the respective stride.

This also requires a new formula.

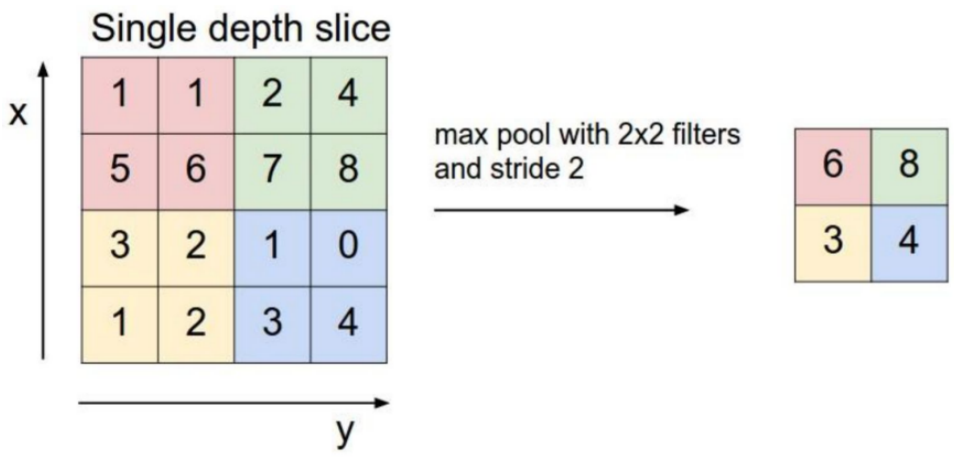



*Figure 21: Max Pooling. Take the max from each stride: R6, G8, Y3, B4*

$$o = \frac{i - k}{s} + 1 \tag{24}$$

### 4.5.2 Average Pooling

Exactly what it sounds like, you take the average of each stride.

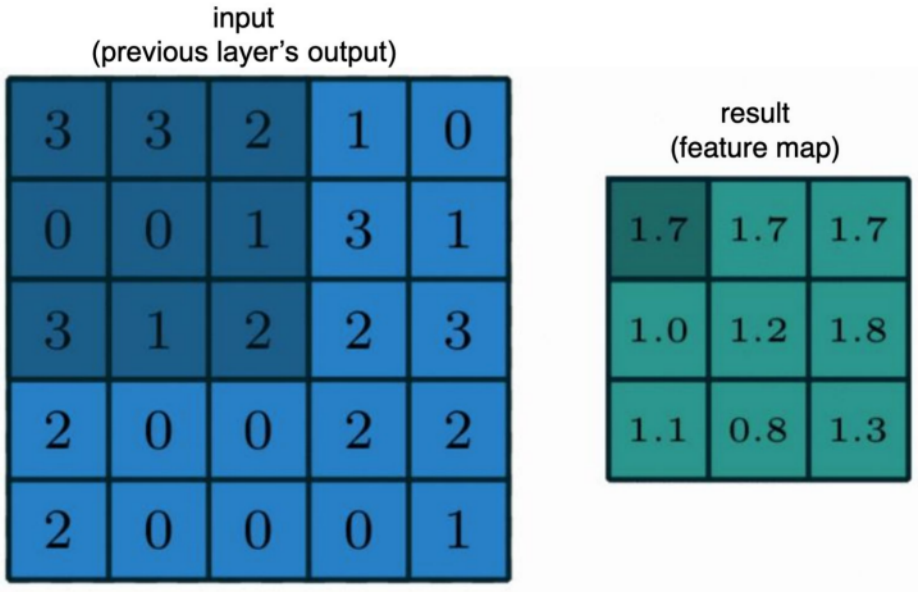



*Figure 22: Average Pooling. Max pooling generally works better*

### 4.5.3 Strided Convolution

Same approach as stride but with pooling operations.

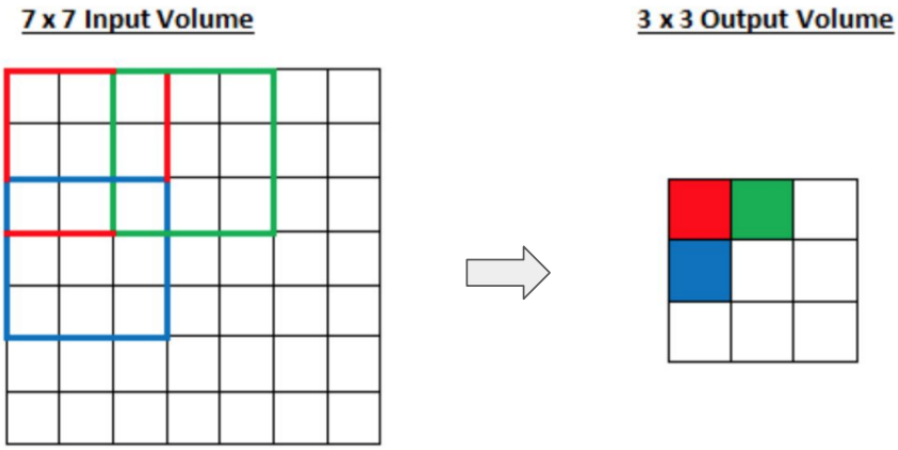



*Figure 23: Strided Pooling*

## ※ 5. Convolutional Neural Network II

### 5.1 Content

This section contains an overview, how we visualize convolutional filters, "CNNs" before the Deep Learning Era, modern architectures, and transfer learning.

### 5.2 Overview

Last section we went over the idea that input data from images are represented by 2-D grids, and if they have colour data, 3-D grids. These grids pass through a set of layers which update the inputs which in turn pass through more layers. These inputs will come out smaller due to downsampling and will go through a classification network where the necessary loss functions are applied. The takeaway is that this whole procedure is end to end, and the goal is to learn how we should update the weights.

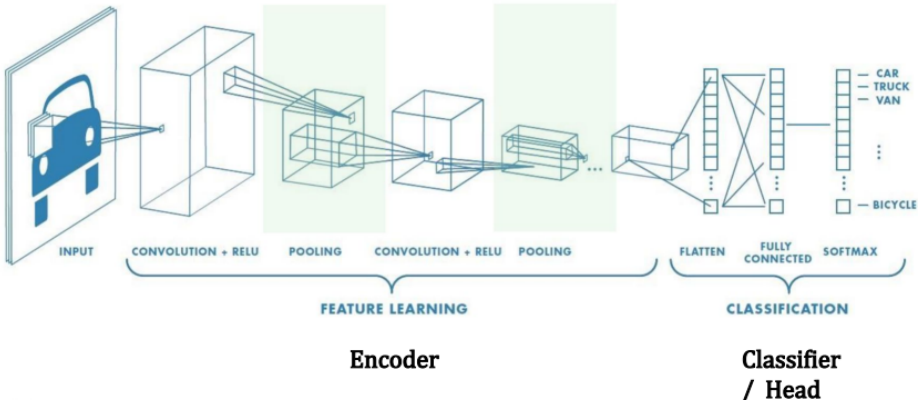



*Figure 24: Typical CNN procedure*

## 5.3 Visualizing Convolutional Filters

With all this talk about filters, what do they actually look like? If we look at Figure 25, each of the filters in Conv1 are applied onto the Input image. In this case, these filters appear as edge detectors and we can see on the channels of Conv1 Output, the different, specific features that are yielded from the filter channels. This is all happening in the first layer.

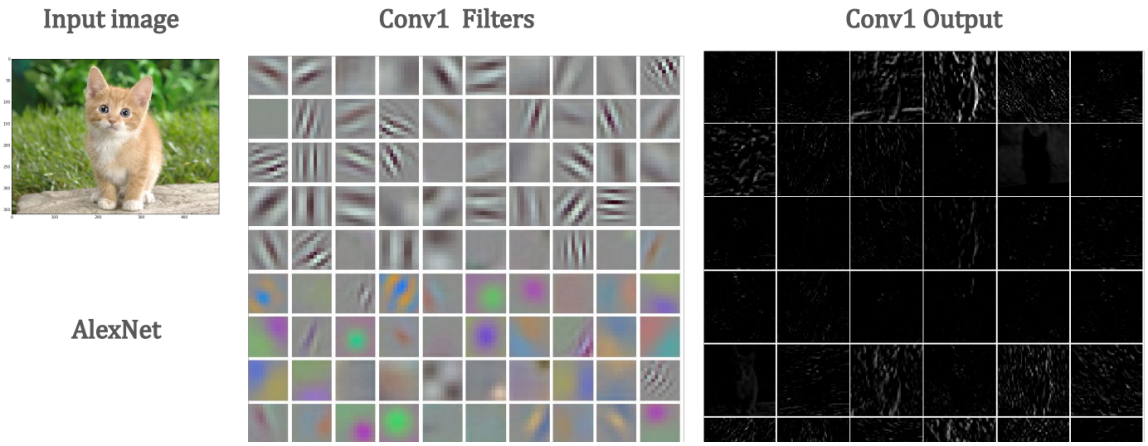



*Figure 25: First layer visual of filtured input*

Figure 26 shows the second layer of the process. The inputs for the Conv2 Filters will be using the outputs from the Conv1 Output. We can see that we apply abstract filters, i.e. "higher level features", and we end up with outputs that seem unusable to humans, but to the model, these outputs are more sophisticated and will help classify whether an image is a cat or not cat.

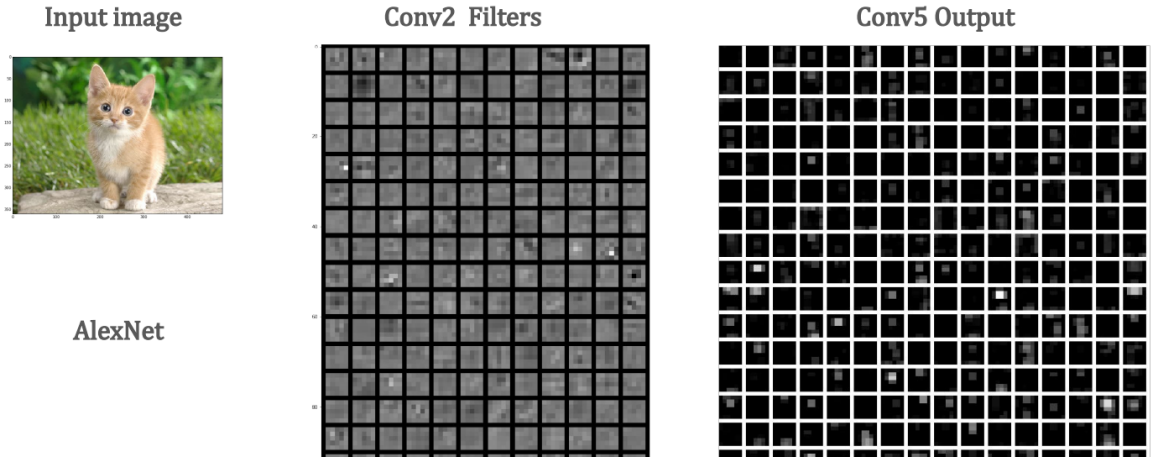



*Figure 26: Second layer visual of filtured input*

### 5.3.1 What features do CNNs learn?

Models will learn to recognize certain features of an image via intuition. The process is as follows.

1. Feed the image to the network
2. Compute the gradients back to the input image
3. Take the maximum value of absolute gradients across channels
4. Visualize

We can then decide whether we want to produce a saliency map, or an image that highlights either the region on which people's eyes focus first or the most relevant regions for machine learning models. Fair warning though, saliency maps are not practically very useful, and can sometimes be misleading.

## 5.4 CNNs in Pre-Deep Learning Era

Originally, CNNs were developed by Yann LeCun, back in 1989. Several variants were developed but CNNs as we know them today started with the LeNet-5. It was a simple CNN with 7 layers: 2 convolutional, 2 pooling, and 3 fully-connected.

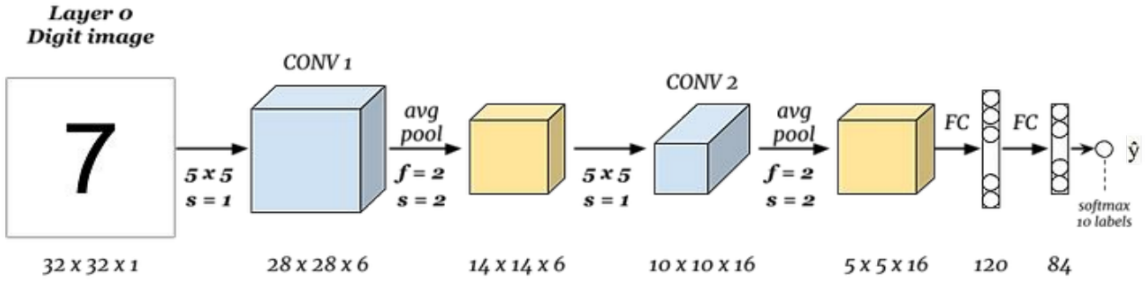



*Figure 27: LeNet-5 CNN*

Figure 28 shows the modern PyTorch implementation of the LeNet-5 CNN. Different components of the model are represented within the code.

The lessons learned from this pre-deep learning era was that Visual Object Classification is holy grail of computer vision and that the best solutions to object classifications were deformable parts models. How do we recognize a face? Try to find face features such as eyes, nose, etc. What's the problem with this idea? We're missing the location of said features.

```
class LeNet5(nn.Module):
    def __init__(self):
        super(LeCun, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(5 * 5 * 16, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 5 * 5 * 16)
        x = F.tanh(self.fc1(x))
        x = F.tanh(self.fc2(x))
        x = self.fc3(x)
        return x
```

*Figure 28: LeNet-5 CNN implementation*

### 5.4.1 Deformable Parts Models

Deformable parts models recognize using parts and locations of parts and we allow some deformation of the part location. However, this is hard to extend to many different types of objects. It also doesn't work well for different viewing angles.

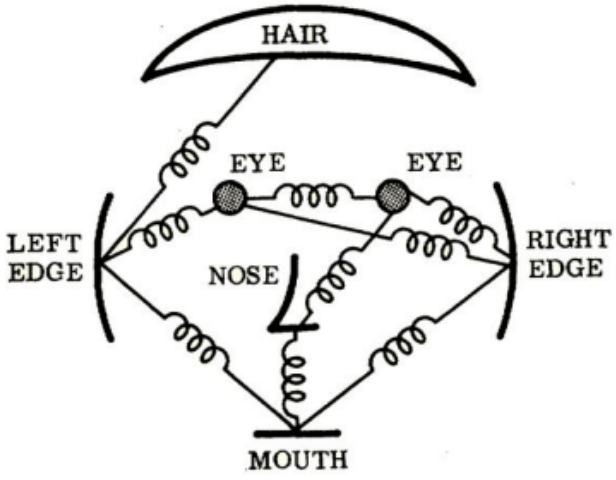



*Figure 29: Model rendition, notice the different parts*

### 5.5 Modern Architectures

## ※ 6. Unsupervised Learning

### 6.1 Content

This section contains a segment on motivation, autoencoders, variational autoencoders, convolutional autoencoders, pre-training with autoencoders, and self-supervised learning.

### 6.2 Motivation

Ask yourself: "What is one thing that all the algorithms covered up to now have had in common?". They all require data with some corresponding ground truth, meaning all the algorithms discussed so far need data that has known, correct outcomes (ground truth) to learn from and make accurate predictions.

Some challenges with supervised learning include the large amounts of labeled data required, the price that can come with acquiring the data, the more common case of unlabeled data than labeled, and the case that the real world doesn't emulate the data exactly.

Here's anothe question to ask yourself: "How do humans learn? How do we learn when we're younger?". We learn by recognizing patterns within observations without explicit supervisory signals, i.e. unsupervised learning. Our brains constantly observe the world for patterns or some sense of structure and these patterns of similar features can tell us a great deal about the data before we can apply a label.

Before we move on, let's establish some definitions.

Unsupervised Learning

- Learning patterns from data without human annotations (unlabeled)

Self-supervised Learning

- Use the success of supervised learning without relying on human provided supervision (automatic supervision)

Semi-supervised Learning

- Learning from data that mostly consists of unlabeled samples, i.e. mix of both labeled and unlabeled

### 6.3 Autoencoders

Autoencoders are unsupervised. If we have an image, we have the data coming in that includes dimension data, colour, etc. We need to find efficient representations of input data that could be used to reconstruct the original input using two components: encoders and decoders. The encoder converts the inputs to an internal representation whereas the decoder converts the internal representation to the output. Encoders do *dimensionality reduction*, decoders are a *generative network*. Below, Figure 30 displays a visual on what these two terms mean.

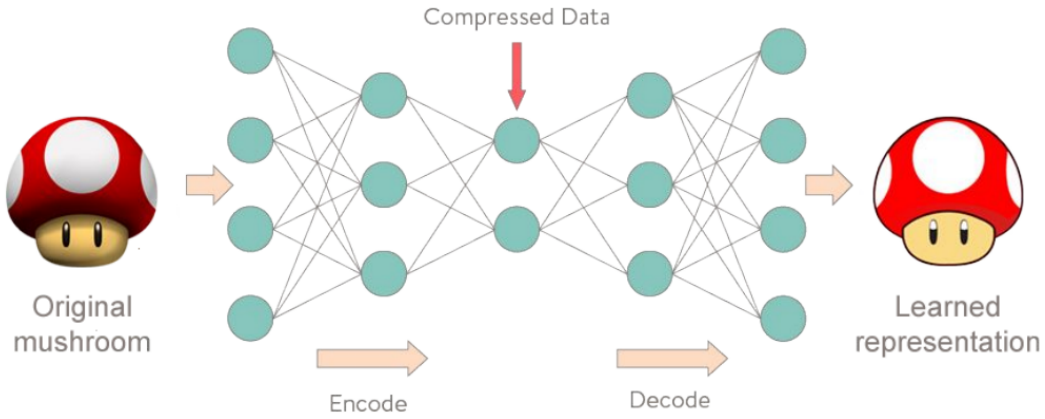



*Figure 30: Hourglass shape creating a bottleneck layer, we dimensionaly reduce then generate the network*

Remember however: **The number of outputs is the same as the number of inputs**. We are forcing the network to learn the **most important features** in the input data and disregard/neglect the unimportant ones. Figure 31 demonstrates the more technical mechanics of the autoencoders.

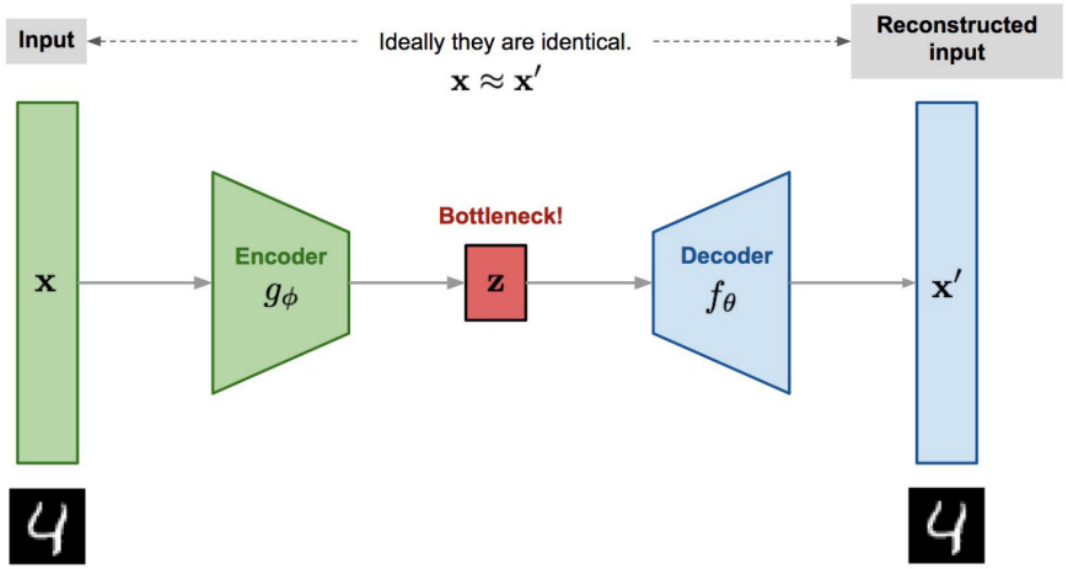



*Figure 31: Technical representation of Figure 30*

Some of the applications of autoencoders include:

- Feature Extraction
- Unsupervised Pre-training
- Dimensionality Reduction
- Generating New Data
- Anomaly/Outlier Detection

### 6.3.1 Stacked Autoencoders

Autoencoders can have multiple hidden layers, these are called stacked (deep) autoencoders. They are typically symmetrical with regards to the central coding layer.

### 6.3.2 Visualizing Reconstructions

One way to ensure that an autoencoder is properly trained is to compare the inputs and the outputs. Let's take a look at Figure 32.

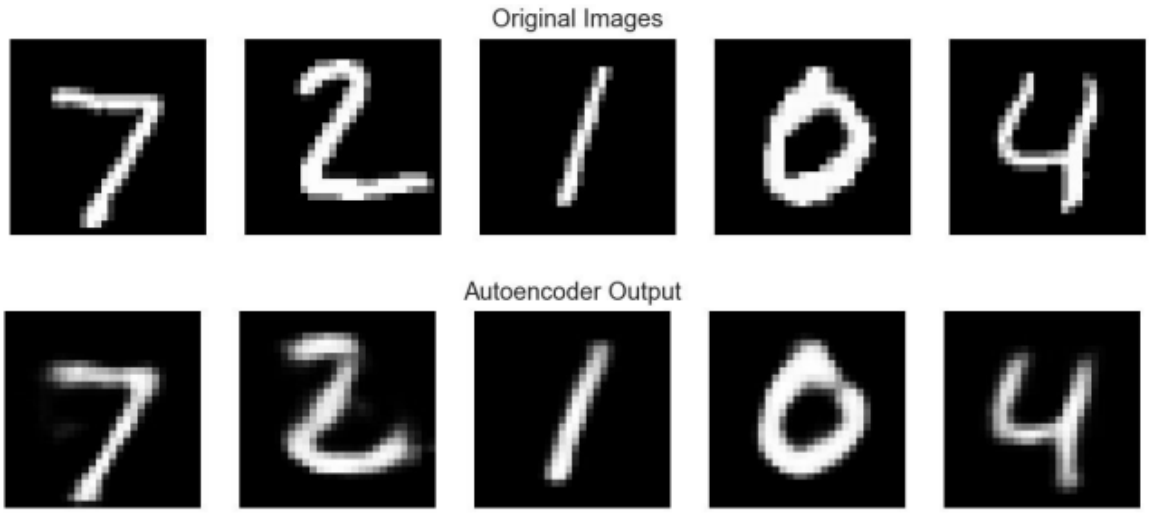



*Figure 32: Output from an autoencoder*

Notice how the output isn't a 1:1 reconstruction of the original image. We can notice some blur lines along the edges and the overall set of outputs aren't as sharp. If we produced a perfect reconstruction, the difference would be $x - x' = 0$ where $x$ is our original images, $x'$ is our output.

### 6.3.3 Denoising Autoencoders

Sometimes we need to make sure that our networks aren't just copying the inputs to its outputs. We can check by applying noise to our input images in order to force it to learn useful and specific features. The steps are as follows:

1. Apply Gaussian noise to input image
2. Pass image through encoder
3. Runs through the code
4. Pass through decoder
5. Produce an output image

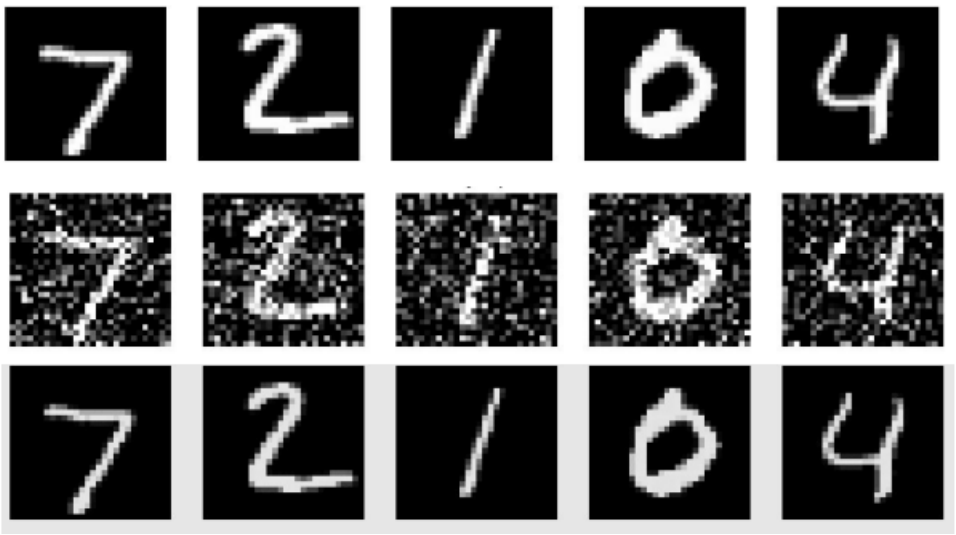


*Figure 33: Applying and removing noise from inputs*

### 6.3.4 Generating New Images with Interpolation

Reducing the image's dimensionality requires structuring the embedding space so that the network maps similar images to similar embeddings. We can employ interpolation to help us.

First compute low-dimensional embeddings of two images. Then interpolate between the two embeddings and decode those as well. The interpolated codings result in new images that are somewhere in between the two original images.

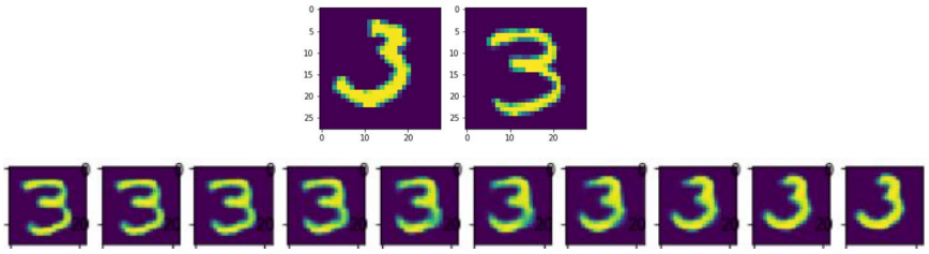


*Figure 34: Interpolation*

### 6.4 Variational AutoEncoders (VAE)

Variational autoencoders are different to the autoencoders we've learned thus far, that their outputs are partly determined by chance even after training and that they can generate new instances that look like they were sampled from the training set. They impose a distribution constraint on the latent space to have a smooth space.

Encoders generates a normal distribution with mean, $\mu$, and a standard deviation, $\sigma$, instead of a fixed embedding. An embedding is sampled from the distribution and decoder decodes the sample to reconstruct the input.

### 6.5 Convolutional Autoencoders

Convolutional autoencoders take advantage of spatial information. Where encoders learn visual embedding using convolutional layers and decoders up-sample the learned visual embedding to match the original size of the image.

#### 6.5.1 Transposed Convolution

The opposite of the convolution is the transposed convolution. They work with filters, kernels, padding, and strides, just like convolution layers. But isntead of mapping $K \times K$ pixels to 1, they can map from 1 pixel to $K \times K$ pixels. The kernels are learned just like normal convolutional kernels.

$$o = (i - 1) \times s + (k - 1) - 2p + op + 1 \tag{25}$$

How do they work? The follow the following process:

1. Take each pixel of your input image
2. Multiply each value of your kernel with the input pixel to get a weighted kernel
3. Insert it in the output to create an image
4. Where the outputs overlap, sum them

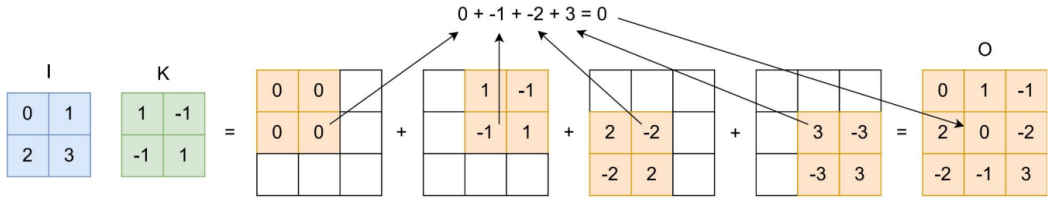



*Figure 35: Transposed convolution*

### 6.5.2 Padding

Opposite process to how padding works in traditional convolutional layers:

1. Compute the output as normal
2. Remove rows and columns around the perimeter

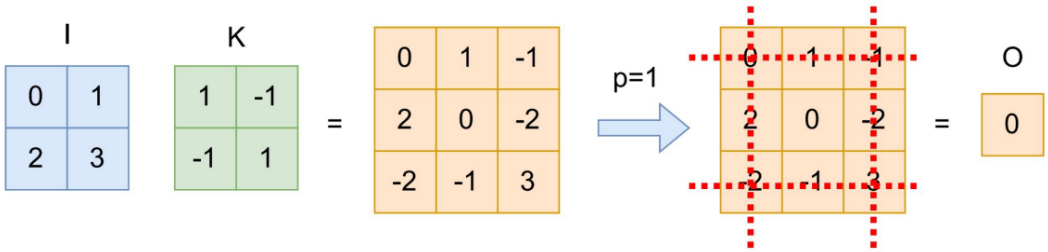



*Figure 36: Padding*

### 6.5.3 Strides

Also the opposite from what happens with traditional convolution layers. Increasing the stride results in an increase in the upsampling effect:

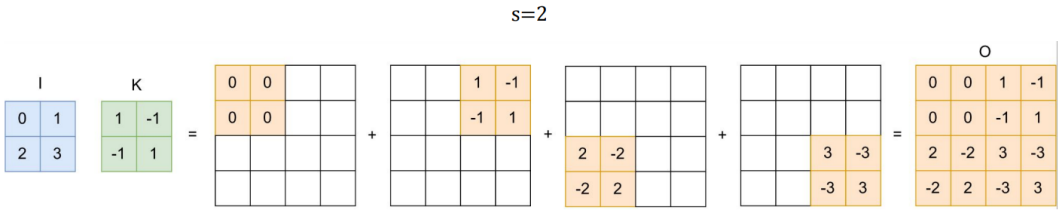



*Figure 37: Strides*

## 6.6 Pre-training with Autoencoders

Previously we discussed how transfer learning could use features obtained from ImageNet data to improve classification on other image tasks. Autoencoders can achieve

similar results by pretraining on large sets of unlabeled data of the same type, only without the labels.

### 6.7 Self-Supervised Learning

What if we could transform unsupervised learning into a supervised setting? The challenge lies in designing tasks that compel the model to learn robust representations.

## ※ 7. Recurrent Neural Network I

### 7.1 Content

This section covers the motivation behind unsupervised learning, word embeddings, distance measures, language models, and RNNs.

### 7.2 Motivation

Autoencoders can be used to learn an embedding space, i.e. encoders conduct data → embedding conversions and vice-versa for decoders. But how can we learn the embedding of words?

#### 7.2.1 Numeric Features

We need to find a way to convert words into numerical features. A way to do this is treat each word as a feature itself, e.g. cat = 1, dog = 0. But when there is no inherent order among the numbers, integer encoding is not enough. Assuming such an order may lead to poor performance.

#### 7.2.2 One-hot Encoding

A better way to convert to numerical features is via one-hot encoding. UofT's categorical feature "Term" can take on three possible values: Fall, Winter, and Summer.

```
Fall -> [1, 0, 0]
Winter -> [0, 1, 0]
Summer -> [0, 0, 1]
```

Dimensions grow with number of words and one-hot encoding assumes each word is completely independent.

## 7.3 Word Embeddings

Obviously words aren't like images they don't contain pixels or image data. They more come down to context, cadence, mannerisms, and patterns in speech. How can we possibly set up an autoencoder to recognize and account for such factors in its training?

Each word has its own index, i.e. if there are 1000 words, there are 1000 features. The meaning behind the word depends on its context so we need to look at the words that appear with the word we're focusing on. This is the same as to how we humans learn how to use the words we learn about.

## 7.4 Distance Measures

In order to talk about which words have similar embeddings, we need to introduce a measure of distance in the embedding space. We can use either **"Euclidean Distance"** or **"Cosine Similarity"**.

## 7.5 Language Models

### 7.5.1 Language Modelling

Defined as "Learning probability distribution over sequences of words".

### 7.5.2 Working with Text

How is working with text more challenging than working with images? We need to account for grammar, spelling, the amount of words the model has to learn, words vs character choice, languages, etc.

This is why we have to employ RNNs.

### 7.5.3 RNNs

RNNs can understand all the factors above and also the sentiment that comes with it. Even though two sentences have the same exact words, their varied order can have two different meanings.

We can take variable-sized sequential input, meaning it can remember things over-time, and update a hidden state based on the previous hidden state and input. We then continue updating the hidden state until we run out of tokens, using the last hidden state as the input to a prediction network.

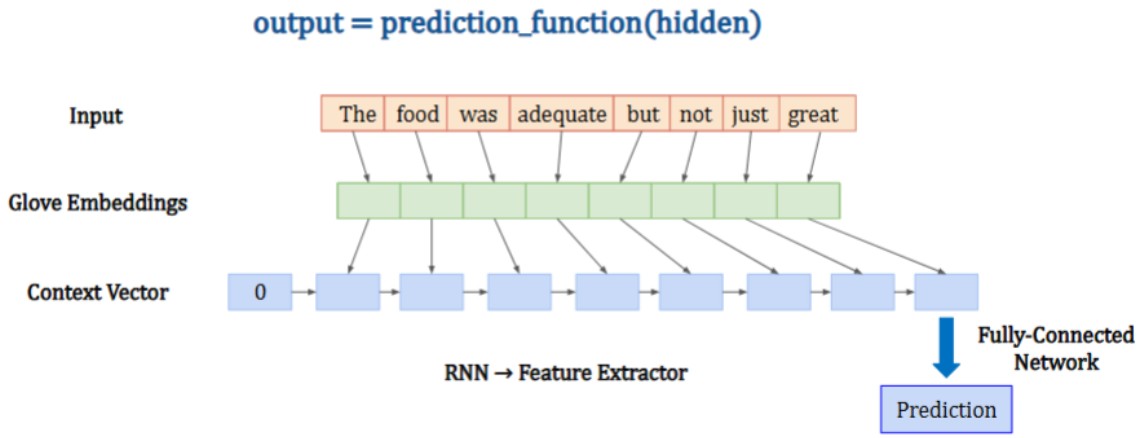



*Figure 38: RNN implementation*

## ※ 8. Recurrent Neural Network II

### 8.1 Content

This section will be an extension of the previous. It goes over the limitations of vanilla RNNS, LSTMs & GRUs, Deep & Bidirectional RNNs, and Sequence-to-Sequence Models.

### 8.2 Limitations of Vanilla RNNs

Many real world problems deal with inputs/outputs with varying sizes. If we consider the concatenated input/hidden and output/hidden vectors as simply input/output, the forward path in the RNN is simply a fully-connected NN.

What happens to RNNs unrolled onto a long sequence? They can become very deep (depth = length of sequence). There are two main problems with vanilla RNNs.

1. They're not good at modelling long-term dependencies
2. They're hard to train due to vanishing/exploding gradients

#### 8.2.1 Exploding/Vanishing Gradients

What are exploding/vanishing gradients? Suppose we have a linear update function. For simplicity, let's ignore inputs.

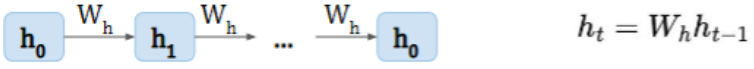



*Figure 39: Our simple linear update function*

We can write this for all time-steps as the following.

$$h_t = (W_h)^t h_0 \tag{26}$$

From here we get two problems:

$$\text{Exploding gradients} \quad h_t \to \infty \qquad \text{if } |W_h| > 1 \tag{27}$$

$$\text{Vanishing gradient} \quad h_t \to 0 \qquad \text{if } |W_h| < 1 \tag{28}$$

### 8.2.2 Tackling Exploding/Vanishing Gradients

To overcome such issues we do conduct the following.

**Gradient clipping → Exploding Gradient**:
If the gradient is greater than a threshold, set the gradient to the threshold.

**Skip-connection → Vanishing Gradient**:
Ideally we want to skip connections to the previous states but this gets expensive. We could instead preserve the hidden state/context over the long term.

## 8.3 LSTMs & GRUs

### 8.3.1 Long Short-Term Memory (LSTM)

LSTMs consist of a long-term memory (cell state) and a short-term memory (context or hidden state). They use three gates to update the memories.

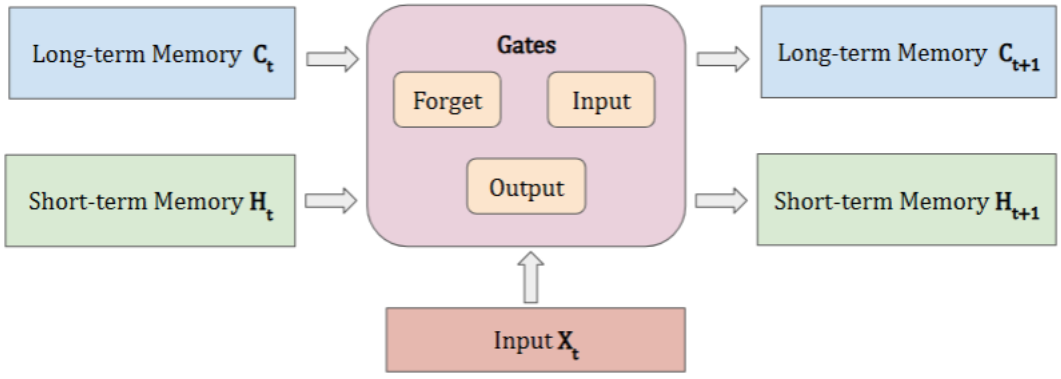
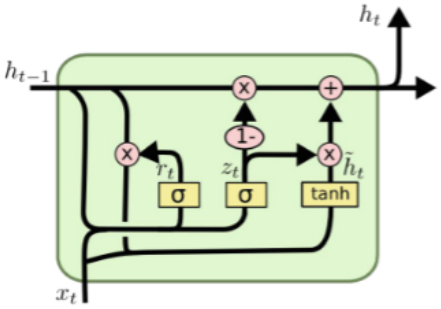



*Figure 40: LSTM sequence*

### 8.3.2 Gated Recurrent Unit (GRU)

GRUs are more efficient than LSTMs while having a similar performance. They Combine forget and input gates into an update gate. They also merge the cell state and hidden state.




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

*Figure 41: GRU gates*

### 8.3.3 LSTM/GRU vs. RNN

LSTMs and GRUs can be trained on longer sequences and are superior at learning long-term relationships compared to vanilla RNNs. They are easier to train, achieve better performance, and with more training, can further improve their convergence and results.

## 8.4 Deep & Bidirectional RNNs

### 8.4.1 Bidirectional RNNs

A typical state in an RNN (RNN, GRU, or LSTM) relies on both the past and present. In tasks like machine translation, where predictions depend on the past, present, and future, exploiting future information can improve performance.

### 8.4.2 Deep RNNs

Stacking RNN layers can help learn more abstract representations; the first layers are better for syntactic tasks, while the last layers perform better on semantic tasks.

## 8.5 Sequence-to-Sequence Models

### 8.5.1 RNN Model Types

We have focused on many-to-one and many-to-many (same length) RNN architectures. However, various tasks, such as translation and image captioning, require slightly dif-

ferent RNN configurations to effectively address their unique requirements.

### 8.5.2 Hidden State Differences

RNNs for prediction (**Encoder**):
- Processes tokens one at a time
- Hidden state represents all the tokens read thus far

RNNs for generating sequences (**Decoder**):
- Generates tokens one at a time
- Hidden state is a representation of all the tokens to be generated

### 8.5.3 Sequence-to-Sequence RNNs

Learning to generate new sequences requires addressing several challenges. One major issue is determining how to generate variable-length sequences and knowing when to stop or finish a generated sequence. Additionally, the training-time behavior must be adjusted through techniques like teacher-forcing, where the model is guided by the correct outputs during training. At inference time, the behavior changes again, requiring methods like sampling and temperature scaling to produce more natural and diverse sequences. These adjustments are crucial for successfully training and deploying sequence generation models.

### 8.5.4 During Training

To determine when to stop or finish a generated sequence, we can use dedicated control symbols to define the Beginning of Sequence (<BOS>) and End of Sequence (<EOS>). For example, in the sequence "<BOS> R I G H T <EOS>", once the RNN generates the <EOS> token, we will know it has completed the sequence.

To define the ground-truth and loss for training an RNN to generate sequences, we use the following method: The RNN is trained to generate a specific sequence from the training set. We start by feeding the RNN with the <BOS> token and compare its prediction with the next character, R, using Cross-Entropy loss. Next, we feed it with R and compare the prediction with I, and so on. Finally, we feed it with T and compare the prediction with the <EOS> token.

### 8.5.5 Teacher Forcing

At each step of training, we compute the loss by comparing the ground-truth tokens with the predicted tokens. To make training more efficient, we use a technique called Teacher Forcing, where the RNN is forced to stay close to the ground-truth sequence.

Instead of using the current prediction as the next input, we pass the ground-truth label. This approach helps guide the RNN more effectively during training.

#### 8.5.6 During Inference

During inference, simply selecting the token with the highest probability, as in a classification problem, is not ideal for generative models. This greedy approach often leads to less diverse and grammatically flawed outputs. To encourage diversity and improve results, we sample from the predicted distributions. We will explore three sampling strategies to address this: Greedy Search, Beam Search, and Softmax Temperature Scaling.

**Greedy Search** selects the token with highest probability as the generated token.

**Beam Search** looks for a sequence of tokens with the highest probability within a window.

**Softmax Temperature Scaling** addresses the issue of over-confidence in neural networks by adjusting the input logits to the softmax function using a temperature parameter. A **low temperature** results in larger logits and more confident predictions, leading to higher quality but less varied samples. Conversely, a **high temperature** produces smaller logits and less confident predictions, resulting in lower quality but more diverse samples.

## ※ 9. Generative Adversarial Networks

### 9.1 Content

This section covers generative models, generative adversarial networks (GANs), the problems with training GANs, and adversarial attacks.

### 9.2 Generative Models

Say we have two tasks at hand on a dataset of tweets. The first task is to identify if a tweet is real or fake, which is a supervised task requiring a discriminative model. This model learns to approximate $p(y|x)$, where $y$ is the label indicating whether the tweet is real or fake, and $x$ is the tweet itself. The second task is to generate a new tweet, which is an unsupervised task requiring a generative model. This model learns to approximate $p(x)$, the distribution of the tweets.

### 9.2.1 Generative Model vs. Discriminative Model

What about generative vs. discriminative models? For **discriminative models**, our input $x$ will pass through our discriminative model (e.g. a classifier) and produce an output truth label. For **generative models**, our input encoding will pass through our generative model (e.g. a variational autoencoder) and produce some output $x_t \in \{x_i\}_{i=1}^{n}$.

### 9.2.2 Generative Learning

Generative learning is an **Unsupervised Learning** task. Although there is a loss function associated with an auxiliary task that we know the answer to, there is no ground truth with respect to the actual task we aim to accomplish. Instead of learning labels for data, we focus on learning the **structure and distribution of the data**.

### 9.2.3 Generative Models

A generative model is used to generate new data by utilizing some input encoding. It learns the underlying patterns and structures within the dataset, allowing it to create new instances that resemble the original data.

**Unconditional Generative Models** only get random noise as inputs and have no control over what category they generate

**Conditional Generative Models** use one-hot encoding of the target category along with random noise, or an embedding generated by another model (e.g. CNN), to provide high-level control over the generated output. This allows us to guide the generation process according to specific categories or features

### 9.2.4 Problem with Autoencoders

Vanilla autoencoders generate blurry images with blurry backgrounds. To minimize the MSE loss, autoencoders predict the average pixel data. Can we use a **better loss function**?

*Figure 42: Blurry image generation*

## 9.3 Generative Adversarial Networks (GANs)

The idea behind GANs is to train two models: the generator and discriminator.

**Generator**: Tries to trick the discriminator by generating real-looking images.
A noise vector → A generated image

**Discriminator**: Tries to distinguish which image is real and fake
An image → A binary label (real/fake)

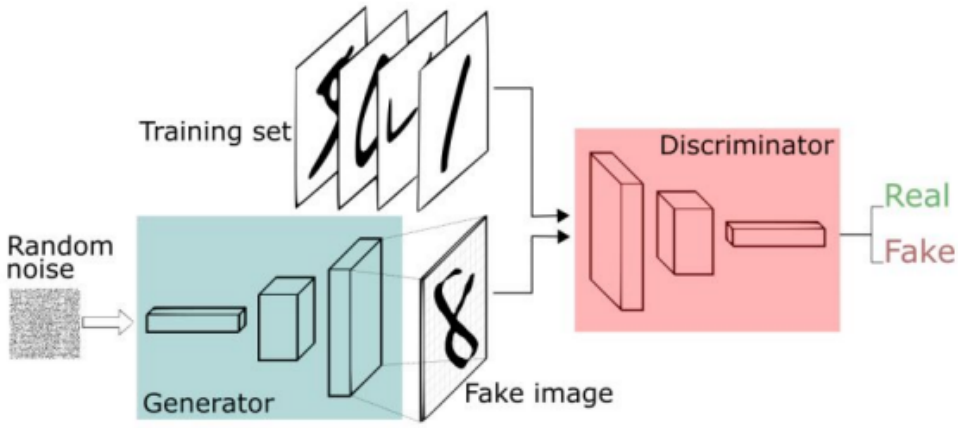



*Figure 43: Generator-Discriminator mechanism*

**Note:** The loss function of the generator is defined by the discriminator

### 9.3.1 Loss Function for MinMax Game

Learn discriminator weights to maximize the probability that it labels a real image as real and a generated image as fake. What loss function should we use? **Binary Cross-Entropy**!

Learn generator weights to maximize the probability that the discriminator labels a generated image as real. What loss function should we use? **Discriminator**!

## 9.4 Problems of Training GANs

### 9.4.1 Vanishing Gradients

If the discriminator is too effective, the generator struggles to learn because small changes in its weights won't change the discriminator output. Remember that we are using the discriminator as a loss function for the generator. If small changes in generator weights make no difference, then we can't incrementally improve the generator (no gradients!).

### 9.4.2 Mode Collapse

Say we want the generator to generate a variety of outputs. If the generator starts producing the same output (or a small set of outputs), the best strategy for the discriminator is to simply reject that output.

However, if the discriminator is trapped in local optimum, it cannot adapt to generator, and the generator can fool it by only generating one type of data (e.g. only digit 1).

### 9.4.3 Failing to Converge

Due to the MinMax optimization process, training Vanilla GANs is very difficult. To train GANs faster, we'll use LeakyReLU Activations instead of ReLU, batch normalization, and regularizing discriminator weights and adding noise to discriminator inputs.

## 9.5 Adversarial Attacks

The goal of adversarial attacks are to choose a small perturbation $\epsilon$ on an image $x$ so that a neural network $f$ misclassifies $x + \epsilon$, i.e. we are setting up the network to fail.

We approach this by using the same optimization process to choose $\epsilon$ to minimize the probability that $f(x + \epsilon)$ = the correct class. We treat $\epsilon$ as the params.

### 9.5.1 Targeted vs. Non-Targeted Attack

**Non-targeted attack**: Minimize the probability that $f(x + \epsilon)$ = correct class.

**Targeted attack**: Maximize the probability that $f(x + \epsilon)$ = target class.

### 9.5.2 White-Box vs. Black-Box Attacks

**White-box attacks**: Assumes that the model is known and we need to know the architectures and weights of $f$ to optimize $\epsilon$.

**Black-box attacks**: We use a differentiable substitute model to optimize $\epsilon$ since we don't know the architectures and weights of $f$. Adversarial attacks typically transfer across models, making this approach viable.

### 9.5.3 Defence Against Adversarial Attacks

It is a very active area of research, and we still don't know how to handle them.

# ※ 10. Transformers

## 10.1 Content

This section covers the motivation behind transformers, the attention mechanism, and transformers.

## 10.2 Motivation

RNNs, Vanilla RNNs, LSTMs, and GRUs are used to capture sequences. But they sucked at capturing long-term dependencies. This was due to the exploding/vanishing gradients that would pop up. The deeper the network, the more sequences were involved, meaning a higher chance of experiencing such gradients. This means they are inefficient as we can't take full advantage of vectorization or our GPU's capabilities.

## 10.3 Attention Mechanism

Attention mechanisms are the building blocks of transformers and they try to emulate how humans think when we see an image and fixate on the smaller details or when we read a text and need to reread the more difficult parts.

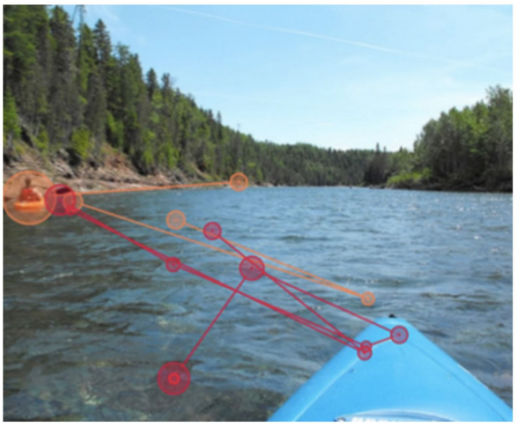


*Figure 44: Our eyes "tunnel-vision" on certain aspects of the image.*

### 10.3.1 Mechanics

Networks using attention mechanisms work by having an **attention score**. This score indicates the importance of different parts of the data. We can organize then aggregate the data according to said score.

### 10.3.2 Simple Attention Mechanism

Let's say we want to make an attention-based pooling for classifying Twitter tweets so we can use it to pool the word embeddings of a tweet into a **tweet embedding**.

We can use a fully-connected network to process word embeddings, generating a **single score for each embedding**. These scores can then be normalized across all words in the tweet using a softmax function.

We then multiply each embedding with its normalized score and sum them all up. This network is trained end-to-end with the classifier.

$$c_i = \sum_j \alpha_{ij} h_j \tag{29}$$

### 10.3.3 Attention Taxonomy

**Cross-Attention:** Between two sequences

**Self-Attention (Inta-Attention):** Compute attention of input with respect to itself: for a given token of the input, compute attention weight for all other tokens in the sequence

### 10.3.4 Computing Attention Score

If we have 2 embeddings, $a, b \in \mathbb{R}^d$, we can use different methods to compute attention score between them.

| | |
|---|---|
| Dot product | $\text{score}(a,b) = a^T . b$ |
| Cosine similarity | $\text{score}(a,b) = a^T . b / \|\|a\|\| . \|\|b\|\|$ |
| Bilinear | $\text{score}(a,b) = a^T W b$ |
| MLP | $\text{score}(a,b) = \text{Sigmoid}(W[a;b])$ |

*Figure 45: Some methods computing attention score*

## 10.4 Transformers

### 10.4.1 Attention in Transformers

Attention in transformers model a neural dictionary. They retrieve a (v)alue, for a (q)uery, based on a (k)ey. These variables are d-dimensional embeddings.

However, rather than retrieving a single value for a query, it uses a soft retrieval where it retrieves all the values but then computes their importance w.r.t query based on the similarity between the query and their keys.

$$\text{attention}(q, k, v) = \sum_i \text{similarity}(q, k_i) \times v_i \tag{30}$$

Such that X is an input in the sequence consisting of n tokens. We use 3 linear layers to compute the queries, keys, and values from the X.

$$\mathbf{Q}=\mathbf{X}\mathbf{W}_Q,\ \mathbf{X}\in\mathbb{R}^{n\times i},\ \mathbf{W}_Q\in\mathbb{R}^{i\times k},\ \mathbf{Q}\in\mathbb{R}^{n\times k}$$
$$\mathbf{K}=\mathbf{X}\mathbf{W}_K,\ \mathbf{X}\in\mathbb{R}^{n\times i},\ \mathbf{W}_K\in\mathbb{R}^{i\times k},\ \mathbf{K}\in\mathbb{R}^{n\times k}$$
$$\mathbf{V}=\mathbf{X}\mathbf{W}_V,\ \mathbf{X}\in\mathbb{R}^{n\times i},\ \mathbf{W}_V\in\mathbb{R}^{i\times v},\ \mathbf{V}\in\mathbb{R}^{n\times v}$$

*Figure 46: 3 linear layers*

Self-attention in transformers is defined as follows.

$$\text{Attention}(Q, K, V) = \sum_i \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V \tag{31}$$

### 10.4.2 Multi-Head Attention

To improve performance:

1. Divide the representation space to h sub-spaces
2. Run parallel linear layers and attentions
3. Concatenate them back to form the original space

### 10.4.3 Transformer Encoders

Each encoder layer consists of:

1. A multi-head self-attention sub-layer
2. A fully-connected sub-layer
3. A residual connection around each of the two sub-layers followed by layer normalization

### 10.4.4 Positional Encoding

The model does not have recurrent or convolutional layers so it doesn't into account the order of sequence. We use positional encoding to make use of the order which allows the model to easily learn to attend by relative positions.

### 10.4.5 Takeaways

**RNN:**

- Struggling with long-range dependencies
- Gradient vanishing and explosion
- Large number of training steps
- Recurrence prevents parallel computation

**Transformer:**

- Facilitate long-range dependencies
- Less likely to have gradient vanishing and explosion problems
- Fewer training steps
- No recurrence, facilitates parallel computation

## ※ 11. Graph Neural Networks

All neural networks are considered special cases of graph nerual networks. Graph nerual networks are just the most general case of neural networks in this course.

### 11.1 Content

This section covers the motivation, deep sets, graphs, GNNs, GCNs, and GANs.

### 11.2 Motivation

The motivation behind GNNs is that they're designed to infer from data that doesn't come in the form of an image or sequence, i.e., they're non-Euclidean. GNNs excel at capturing complex, irregular relationships in data that is structured as graphs, where nodes represent entities and edges represent the connections between them.

### 11.3 Deep Sets

What happens if we omit the Positional Encoding from transformers?

- The input will be treated as a set and the learned representation won't change if you randomly shuffle the input tokens

- This property is known as order-invariance or permutation-invariance
- But when is this property useful?

Omitting positional encoding from transformers makes the model order-invariant, treating the input as a set rather than a sequence. This is advantageous in contexts where the order of elements is irrelevant, such as set-based problems, graph models, or multi-instance learning. However, in tasks where the sequence order is crucial, like language modeling or machine translation, positional encoding is vital for capturing the correct relationships between tokens.

The model must be invariant to the order of the items as the order is a spurious feature and can mislead the model (similar to why we shuffle the train data).

$$\mathbf{X} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} x_3 \\ x_5 \\ x_1 \\ x_2 \\ x_4 \end{bmatrix} = \begin{bmatrix} x_5 \\ x_3 \\ x_4 \\ x_2 \\ x_1 \end{bmatrix} = \ldots = \begin{bmatrix} x_4 \\ x_1 \\ x_2 \\ x_5 \\ x_3 \end{bmatrix}$$

$$f\left(\{\mathbf{x}_5, \mathbf{x}_1, \mathbf{x}_4, \mathbf{x}_2, \mathbf{x}_3\}\right) = \mathbf{y} = f\left(\{\mathbf{x}_2, \mathbf{x}_5, \mathbf{x}_4, \mathbf{x}_1, \mathbf{x}_3\}\right)$$

*Figure 47: X here can equal all the sets as the order of the inputs don't matter*

1. Learn embeddings for each item → Use a shared neural network $\psi$ (e.g., MLP, Transformer, etc.) to project each item to a shared space

$$\mathbf{h}_i = \psi(\mathbf{x}_i)$$

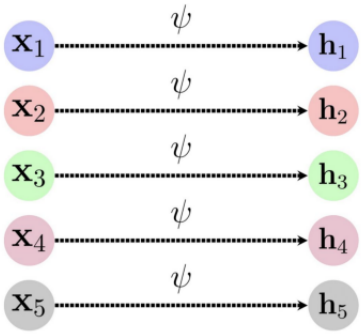



*Figure 48: Each input $x_i$ has a corresponding embed*

2. Learn embeddings for the set → Use an order-invariant aggregation function such as sum, mean, max, etc. to aggregate the embeddings into a single embedding

$$\sum_{i \in \mathcal{V}} \underset{\mathbf{X}}{\psi(\mathbf{x}_i)}$$

*Figure 49: Sum into single embedding*

3. Use another neural network $\phi$ (e.g., MLP) to project the embedding to the final space

## 11.4 Graphs

Recall, transformers can create a fully-connected graph over the input and learn the edge weights. A graph $G = (V, E, X)$ is a data-structure that encodes pair-wise interactions or relations among concepts and objects. Where

- $V$ is the set of nodes representing concepts or objects
- $E \subseteq V \times V$ is a set of edges connecting nodes and representing relations or interactions among them
- $X$ encodes the node features of each node

We can represent the edges in an adjacency matrix A:

$$a_{ij} = \begin{cases} 1, & \text{if } (i,j) \in \mathcal{E} \\ 0, & \text{if } (i,j) \notin \mathcal{E} \end{cases}$$

Degree of a node is number of edges connecting to that node:

$$d(i) = \sum_j a_{ij}$$

## 11.5 Graph Neural Networks (GNNs)

GNNs are neural networks that function on graphs, based on Message-Passing, i.e. communicating with neighbors to update embeddings. We can use them to

- predict node classes
- predict graph classes
- predict links between nodes

### 11.5.1 Message-Passing

For each node in a graph

1. Aggregate embeddings of its neighbor nodes
2. Combine the aggregated embedding with the node embedding
3. Update the node embedding

Aggregates all node embeddings into a graph embedding and must be an order invariant function such as sum, mean, max, attention, etc. Different instantiations of aggregation function define different GNN models.

## 11.6 Graph Convolutional Networks (GCNs)

A layer of a GNN is basically a nonlinear function over node features and adjacency matrix.

$$H = f(A, X) \tag{32}$$

The simplest model that we can define being

$$H = \sigma(AXW) \tag{33}$$

Where $\sigma$ is the non-linearity and $W$ is the weight matrix. This is already a strong model, but it has two limitations.

1. Multiplication with $A$ means that, for every node, we sum up all the feature vectors of all neighboring nodes but not the node itself
   **Fix:** Add self-loops (add the identity matrix to $A$) $\longrightarrow A = A + I$
2. $A$ is not normalized and therefore the multiplication with $A$ will completely change the scale of the feature vectors
   **Fix:** Symmetrically normalize $A$ using diagonal degree matrix $D$ such that all rows sum to one $\longrightarrow D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$

With these fixes, the GCN layer is defined as follows:

$$H = \sigma(D^{-\frac{1}{2}}AD^{-\frac{1}{2}}XW) \tag{34}$$

### 11.6.1 Going deeper with GNNs

A GCN layer updates node embeddings by incorporating features from immediate neighbors (through multiplication with the adjacency matrix, $A$). By stacking multiple GCN layers, we extend the influence to nodes further away in the graph, similar to how increasing the number of layers in CNNs expands the receptive field.