6.S191 Introduction to Deep Learning

Thanadol Chomphoochan
IAP 2021

1 Introduction to Deep Learning

1.1 Logistics

The first lecture was given by Alexander A. Amini on January 18th, 2021. He started by telling the class that he would introduce us to 6.S191 through the same method he already used in previous year. If you know what happened in the previous year, it is pretty amazing: Barack Obama recorded a video to introduce us to 6.S191! Of course, that was actually a fake video generated by an AI. The audio shown in class this year was noticeably degraded, however. Amini explained that this was done deliberately to prevent any potential misuses, which is understandable given the class is on Zoom and anybody could steal the video. Even then, the generated images were quite impressive.

What is deep learning? **Deep Learning** is "extracting patterns from data using neural networks." This is a subset of **Machine Learning** which means, more generally, a way to learn something without us having to hard-code the steps. The broadest characterization is **Artificial Intelligence**, which is "any technique that enables computers to mimic human behavior." For the first lecture, we will focus on the neural networks, the foundation of deep learning. Later on, we will discuss a lot of interesting topics. For this year specifically, there are new exciting topics like probabilities and algorithmic biases. Then, we conclude with guest lectures and the final project.

There are two options to fulfill the final class project requirement. The first option is to work in teams of up to four members to develop a cool deep learning idea and write a proposal. The ideas will be judged not by novelty but how well thought out they are respective to the class contents. After all, this is only a two-week class. A live presentation of the final project is also required. Only three minutes is allowed. As Amini described, presenting a beautiful idea concisely is an art. Top winners, determined by a panel of judges, can win various prizes like NVIDIA 3080 GPU, Google Home Max, and large display monitors. The other, less intensive option is to write a 1-page review of a deep learning paper, which will be graded on clarity. (The class is graded on a P/D/F basis.)

After each lecture, there are "software labs" for us to practice implementing the ideas in deep learning. There are also prizes for students who complete each lab well. The prizes are Beats headphones, 24" HD display monitor, and a quadcopter drone, for lab 1, lab 2, and lab 3 respectively. The software labs are not mandatory but they are very useful for learning. Office hours, specifically for help on software labs, are held in Gather.Town.

1.2 Why Deep Learning and Why Now?

Hand engineered features are time-consuming, brittle, and not scalable in practice. Deep learning will try to learn these features from the data in a hierarchical order. For example, to detect faces, we might try to learn the low level features such as lines and edges first, then the mid level features such as eyes and nose, and then the high level features would distinguish facial structures. Of course, the neural networks might not work exactly like this, but the idea of having multiple layers of features is crucial to understanding what is going on.

We study deep learning now because data has become "much more pervasive." The algorithms have been developed to a point where they can be scaled and parallelized on GPUs (i.e. They run very quickly!). Software tools and libraries have become much more developed as well.

1.3 The Perceptron

We consider the most basic structure in deep learning: the **Perceptron**. A perceptron takes in a single input data and then output a value. This value is generated by first taking a linear combination of features in the input data and then applying an **activation function**. For a single input point, the perceptron can be expressed with the following formula:

$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} x_i w_i\right)$$

The input is denoted as x which has m features, x_1 through x_m. The weights for taking a linear combination are denoted w_i. A bias term, w_0, is also introduced to shift the value before we pass it into g, a non-linear activation function. The output is $\hat{y}$.

To express this more concisely, we put all features into a column vector we denote X. The weights can also be put in a vector W. Thus, the perceptron can be expressed as follows:

$$\hat{y} = g\left(w_0 + X^\top W\right).$$

Activation functions people commonly use as g include the sigmoid function, hyperbolic tangent, or rectified linear unit (ReLU).

We need these activation functions because we need to introduce non-linearity into the network. Why? Let us take for example the case where $w_0 = 1$ and $W = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$. If g is the identity function, namely $g(x) = x$, then we will basically have the line $1 + 3x_1 - 2x_2 = 0$ as the decision boundary. For any input $X = \begin{bmatrix} y \\ z \end{bmatrix}$, it will be classified as either positive or negative depending on which side of the line it lines on, but that is about it. Real life data might not be separable with just a single line. Without these functions, we can only have a single straight line as our decision boundary, which is almost never enough.

A perceptron can also output multiple values. First, we define the pre-activation z_i:

$$z_i = w_{0i} + \sum_{j=1}^{m} x_j w_{ji}.$$

Then, the output y_i is simply $y_i = g(z_i)$.

1.4 Constructing a neural network

Now, we can construct a simple neural network which has one **hidden layer**. Put simply, we will have a layer of a multi-output perceptron that converts inputs into some number of values. Then, we use these values to feed into the final output layer which is also a multi-output perceptron. Because all inputs are densely connected to all outputs, these hidden layer and output layer are called **dense** layers. Tensorflow already implemented the dense layers for us, so we can simply call the function. We can also have neural networks with multiple layers, called a **deep neural network**, simply by specifying the size of each layer sequentially.

1.5 Quantifying loss

Consider the problem of deciding whether students will pass the class, 6.S191. Suppose we have two features: The number of lectures a student attends denoted x_1 and the number of hours spent on the final project denoted x_2. We can plot each student on an x_1–x_2 graph, labeling each data point with "pass" or "fail."

Given a new data point with unknown label, deep learning will try to predict whether we pass the course. Let's say our input is $x^{(1)} = \begin{bmatrix} 4 \\ 5 \end{bmatrix}$. The neural network might predict a value of $\hat{y}_1 = 0.1$ when the actual value is $y_1 = 1$. This error is called the **loss**.

On a single data point, we denote the loss as $\mathcal{L}\left(f\left(x^{(i)}; W\right) y^{(i)}\right)$. Note that the loss depends on the weights, W. Taking the mean of losses over all training data we have, we get the **empirical loss**:

$$J(W) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}\left(f\left(x^{(i)}; W\right) y^{(i)}\right).$$

There are many loss functions we can choose to use. For problems where the output is binary (e.g. yes or no), we usually use the **cross entropy loss**. For problems where the output is a real number (a regression problem), we usually use the **mean squared error loss**.

1.6 Training neural networks

The goal of deep learning is to find weights W^* that achieve the lowest loss:

$$W^* = \underset{W}{\operatorname{argmin}}\, J(W).$$

Gradient descent is one way to do this. It first initializes the weights randomly, then in each iteration, it computes the gradient $\frac{\partial J(W)}{\partial W}$ and update the weights by $W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$ where η is the learning rate. Doing this calculation repeatedly allows us to converge to a nearby minimum.

To compute the gradient, we use an idea called **backpropagation**. Suppose we have a simple neural network with just one input feature, one output, and one hidden layer with just one node. The gradient $\frac{\partial J(W)}{\partial w_2}$ describes how much the loss changes with respect to a small change in w_2. We can compute this using the chain rule:

$$\frac{\partial J(W)}{\partial w_2} = \frac{\partial J(W)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_2}.$$

Meanwhile, for w_1, the calculation would be:

$$\frac{\partial J(W)}{\partial w_1} = \frac{\partial J(W)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_1} \frac{\partial z_1}{\partial w_1}.$$

It is possible to write a formula that efficiently "propagates" the gradient from the output layer back to the input layer, hence the name "backpropagation." Of course, we almost never need to figure out the formula ourselves because libraries such as TensorFlow provide a way to automatically calculate the gradients.

1.7 Gradient descent in practice

In practice, training neural networks is difficult. The loss function is usually non-convex. Gradient descent might get stuck at a local optimum. Even setting the learning rate, η, can be very difficult. If we set it too low, we may convert to a local minimum only. If we set it too high, we might overshoot and never converge.

A popular idea to deal with this problem is to try a lot of different learning rates and see what works. Another smarter idea is to design an adaptive learning rate that adapts to the landscape. The algorithms for determining such learning rates include SGD, Adam, Adadelta, Adagrad, RMSprop, and many more.

It is usually infeasible to compute the gradient repeatedly as that requires looping through all input data points. **Stochastic gradient descent (SGD)** fixes this problem by randomly picking

a single data point and estimate the gradient based on that data point only. This works very quickly but is very "noisy." The middle ground is to use batches of B data points, called the **mini-batches**. So,

$$\frac{\partial J(W)}{\partial W} = \frac{1}{B}\sum_{k=1}^{B}\frac{\partial J_k(W)}{\partial W}.$$

This allows for a smoother convergence and allows for larger learning rates. Mini-batches also give us the ability to parallelize computations using GPUs.

1.8 Overfitting and Regularization

This is, Amini claimed, "the most fundamental problem in deep learning." Overfitting is when our model tries to capture noises in our data and does not generalize well to other unseen data. In contrast, underfitting is when our model cannot capture the essence of the data well.

Regularization is a technique that constraints our optimization problem to discourage complex models. This prevents overfitting.

One idea of regularization is called **dropout** which randomly "drop" some percents of the activations to zero. This forces the network to not rely on any one node.

Second regularization technique is called **early stopping**. This basically means we stop our training before the model overfits. To do this, we split our data into two sets: training data and test data. We keep training, reducing our training and testing loss, until we get to the point where the testing loss starts increasing. Of course, the training data will keep decreasing if our model can memorize the data it sees, but this signals to us we should stop because we are overfitting.

2 Deep Sequence Modeling

The second lecture was given by Ava P. Soleimany on January 19th, 2021.

2.1 Sequence modeling applications

We first started by discussing the following problem: Suppose we have an image of a ball and we would like to predict what its next position is. This would be a difficult task because we barely have any information about the ball's movement at all. However, when given a sequence of the ball's positions at each point in time, we can make a sensible prediction by analyzing the trajectory. This is a type of problem called **sequence modeling**, where we are given sequences of letters, words, images, or some other inputs and we have to compute the next logical output.

There are four main types of problems in deep learning. First, a **one-to-one** problem where we are given an input (with many features) and we are supposed to output one thing associated with that input such as a real value. In previous lecture, we discussed the **feed-forward model** which solves this problem by having each layer feed its output into the next layer as an input. The "Will I pass this class?" problem, specifically, is a **binary classification** problem, while other problems with real-valued outputs are called **regression** problems.

The next type is a **many-to-one** problem. One example is the sentiment classification problem where we have a lot of information in sequence (e.g. words) and we try to evaluate the overall sentiment. We also have **one-to-many**, such as the image captioning problem as we only receive one image and have to generate a string of words forming the image description. Lastly, **many-to-many** problems take in a sequence of inputs and then output a sequence of associated outputs. Machine translation is a good example.

2.2 Recurrent Neural Networks (RNNs)

Our **feed-forward neural network** from previous lecture has no notion of "time" at all. Consider a perceptron. A perceptron takes in m features, $x^{(1)}$ through $x^{(m)}$, computes the linear combination to get a pre-activation z, then passes it into a non-linear activation function to get the output $\hat{y} = g(z)$. These perceptrons are densely linked in layers to form a fully-connected neural network. If we want to use the network to perform sequence modeling, one way to do this is to think about an input from each time step as one distinct input then feed it into the neural network to get the corresponding output at that time. Mathematically, if the input at time t is denoted as $x_t \in \mathbb{R}^m$, our output $\hat{y}_t \in \mathbb{R}^n$ is a function of x_t. Intuitively, we know this is not how sequences work. The output should somehow depend on inputs from time $t-1$ and before too, namely, $\hat{y}_t$ is a function of x_t and h_{t-1} where h_{t-1} is some vector that encapsulates the information about the past.

We can address this by allowing our neural network to output a **cell state** h_t alongside the usual output $\hat{y}_t$. This will be passed alongside the input x_{t+1} of the next time step and affects how subsequent cell states and outputs look like. Then, deep learning allows us to "learn" how to construct the cell states from their previous state and current input and how to construct the outputs from the cell states. Namely, given the input vector x_t, we compute the cell state h_t and output $\hat{y}_t$ as follows:

$$h_t = \tanh\left(W_{hh}^\top h_{t-1} + W_{xh}^\top x_t\right)$$

$$\hat{y}_t = W_{hy}^\top h_t.$$

The weight matrices W_{hh}, W_{xh}, W_{hy} are learned. Note that we are using the same weight matrices W in every time step. The neural network with these weight matrices is called a **recurrent cell** as it is used repeatedly and saves its own state to influence the next rounds of computation.

Now, let us convince ourselves this network architecture works by confirming the criteria for sequence modeling problems. To model sequences, we need to be able to do the following:

1. Handle variable-length sequences.
2. Track long-term dependencies.
3. Maintain information about order.
4. Share parameters across the sequence.

First, we consider the issue of how to use sequence modeling for problems which do not involve numbers. Consider the problem "predict the next word." As an example, if the input sequence is "This morning I took my cat for a," then a reasonable output would be "walk." Before we can even begin to train a neural network, we need to somehow represent words as vectors of numerical inputs. We use an idea called **embedding** which transforms indexes into a vector of fixed size. This is done by mapping all words we know in our language to unique numbers, then mapping these numbers to a vector using some techniques. One way to do this is through **one-hot encoding** which creates a vector of dimension equal to the number of words in the language. Each index's corresponding vector is the one with zeroes in all positions but one at the index's value. Index 2, for example, would have the associated vector be $\begin{bmatrix}0 & 1 & 0 & 0 & \dots & 0\end{bmatrix}^\top$. Another technique is **learned embedding** which utilizes deep neural networks to assign words that are similar to each other to vectors that are close together in the hyperspace.

Now that this issue is dealt with, we can see that RNNs satisfy all four conditions. RNNs can handle variable-length sequences because we simply need to change the number of times the recurrent cell is used. Long-term dependencies are implicitly handled through weight matrices which try to save information as cell states. Since cell states depend on the history of input and their interactions, even if we have sequences of same items but in different order, our RNNs would be able to pick up on that and compute differently. Lastly, the weight matrices are shared among all time steps.

2.3 Backpropagation through time

First, we do a forward pass to compute the cell states and outputs for each time step. The backward pass then computes the gradients. Unlike fully-connected neural networks, the backward pass not only propagates the gradient from the loss function to the inputs but also takes into account interactions between each time step to the previous.

One big issue of RNNs has to do with the behavior of the gradients. Suppose we want to calculate the gradient of loss with respect to h_0. We will likely have to multiply a lot of factors starting from the final time step to the initialization. If there are many values greater than one, our gradient will grows very quickly; we call this **exploding gradients**. If there are many values less than one, then we have **vanishing gradients** instead. There are many proposed solutions to these problems. For example, with exploding gradients, we can do **gradient clipping**: scale down the values that are too large. For vanishing gradients, changing the activation function might help. ReLU, for instance, prevents f' from shrinking the gradients when $x > 0$. Initializing the weights to identity matrix has also helped in some cases. Even then, these solutions do not fully solve the problem.

2.4 Long Short Term Memory (LSTM)

We introduce the "gated cells" which help us better track long-term dependencies without vanishing gradients. It can be described as a recurrent cell which selectively lets some information pass through. (Of course, deep learning will help us learn what information to keep without having to put in manual effort.)

The key idea is that information can be added or removed through structures called **gates**. Consider a sigmoid layer followed by coordinate-wise multiplication for example. Recall that sigmoid behaves almost like a piecewise function with value 0 on one side and 1 on the other side. The final result is like all-or-nothing, depending on whether the input is selected through the sigmoid function or not.

I will not go into the details of how LSTM works because there are so many components. Some notable differences from our previous recurrent cell is that an LSTM cell outputs an extra cell state c_t alongside the old h_t and that it uses many different activation functions, including sigmoid and hyperbolic tangent. LSTMs basically work in four steps: forget, store, update, and output. This gating mechanism allows uninterrupted gradient flow through c_t values.

2.5 RNN applications

Soleimany walked us through many different applications of recurrent neural networks.

The first example is music generation, which we will cover in the first lab problem. In this problem, we are given a sheet music we are to output the next note in the music. Of course, we can feed the next note in as an input and keep generating more music as needed. There is a fun historical story about this too: Schubert's work dubbed the "unfinished symphony" was originally intended to be four movements long. However, Schubert passed away before he could finish, so researchers trained an RNN on Schubert's body of work then used it to generate the last two movements of the sympony. Another example task is sentiment classification, which for some reason people love practicing on tweets.

We discussed the machine translation problem in a bit more details. We cannot simply use a simple RNN to generate an output corresponding to each input time step because some languages order the information differently. So, one idea is to use an RNN as an **encoder network** to encode the information from the original language, then pass that vector representation into a **decoder network** which translates to the desired language. This method works because the information representation is independent of the language it is expressed in. However, we have an issue of

encoding bottleneck. If the vector representation is small, then only so much information from the original text can be stored at once.

The recurrent neural networks we learned about today are not parallelizable and thus are very slow in practice. We need to do backpropagation through time in iterations. Tradtional RNNs also have relatively short memory. Even with LSTMs, we cannot handle too long inputs. Researchers have come up with a new method called **Attention** which is becoming more popular. Instead of having the decoder access the final state vector only, the decoder has access to the state outputs of each time step. Attention basically provides learnable memory access.

Two last example applications of RNNs include trajectory prediction for self-driving cars and environmental modeling.

3 Deep Computer Vision

The third lecture was given by Alexander A. Amini on January 20th, 2021. He said Deep Computer Vision is his most favorite topic in 6.S191.

3.1 The rise and impact of computer vision

What is vision? That might be difficult to define explicitly, but at least we can define its purposes:

- To know what is where by looking
- To discover from images what is present in the world, where things are, what actions are taking place, to predict and anticipate events in the world

There are a lot of useful applications if we are able to make the computer "see" as humans do. Some of the examples given include facial recognition, self-driving cars, medicine, biology, healthcare, and accessibility. In class, Amini showed us a rather sentimental video of how deep learning is used to help guide blind people to run independently via audio cues. It is so easy for humans to take vision for granted, but it is not even clear what it would mean for a computer to "see." After all, computers only have access to the pixels which are simply grids of numbers describing the color and brightness of each point.

To tackle the interesting issue of computer vision, our intuition leads us to focus on the idea of high-level feature detection. For instance, dealing with human faces, we want to be able to identify nose, eyes, and mouth. Dealing with cars, we might identify wheels, license plate, or headlights.

One way to do this is manual feature extraction dependent on our domain-specific knowledge. The main problem to this approach is that there are a lot of variations humans cannot account for. For example, there may be scale variation (small vs. large) or viewpoint variation (in 3-D). We need to be able to define those features clearly and come up with the detection system that is sensitive enough to detect those features invariant of minor modifications while being specific enough to filter out other unrelated properties. Obviously, the model is very brittle. Deep learning allows us to learn those features without any manual effort.

3.2 Convolutions

We go back to the neural network we have discussed since the first layer. A neuron in a layer is connected to **all** neurons in the next layer. If we want to do deep learning on a 2-D image, we might have each input feature represent the value of each pixel. So, for a $h \times w$ image, we will have hw input features. Then, we can use this as an input to a fully-connected neural network. Observe that this does not preserve any spatial information at all.

Instead of densely connecting the neurons, we only connect patches of input (i.e. a square region) into each neuron in the layer. For example, the first neuron might represent the top left 3×3 area, then the next neuron represents the same area but shifted to the right once. This

creates a notion of spatial arrangement as each neuron is only affected by the corresponding region. If we have neurons sliding windows of patches on the image, then each neuron would allow us to identify some specific features in the corresponding area.

This idea leads to a **convolution** operation. Given a filter, which is a grid of weights, we can convolve this filter with the image grid in a sliding window manner to get the intensity of a feature for each patch. Note that this means the same set of weights are shared among different patches in the picture. If we want to detect multiple features, we can simply use different filters.

For example, the following 3×3 filter detects a top-left to bottom-right diagonal line:

$$\begin{bmatrix} 1 & -1 & -1 \\ -1 & 1 & -1 \\ -1 & -1 & 1 \end{bmatrix}.$$

If we multiply coordinate-wise with a 3×3 image grid, a higher value would correspond to identifying a diagonal line.

The convolution operation slides a filter over the input image, do element-wise multiplication, then add the outputs. For example, we might have:

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 4 & 3 & 4 \\ 2 & 4 & 3 \\ 2 & 3 & 4 \end{bmatrix}.$$

Using different filters, we can detect different effects. Historically, humans have engineered a lot of filters to do various kinds of effects. Though, deep learning can also help learn which weights to use.

3.3 Convolutional Neural Networks (CNNs)

A **Convolutional Neural Network (CNN)** is similar to a fully-connected neural network but instead of using dense layers, we use layers of convolutions with multiple filters. This mimics the process of trying to identify lower-level features first in the first few layers, and then combining them to get higher-level descriptions that help us solve the desired problem.

Note that we also need non-linearity like our usual NNs. ReLU is commonly used as an activation function. Since images can be very large, we also require a **pooling** mechanism which helps **downsamples** the image. This reduces the dimensionality of the image but still preserves spatial invariance. Oftentimes, people use **maxpooling** which simply finds the max value of features in the sliding window.

Convolutional neural networks are generally used as a part of a bigger network to solve the problem rather than as a standalone solution. Take classification problem for example. First, we use multiple layers of convolution and ReLU as described earlier to learn the high-level features. Then, we flatten those features into a vector of input features to feed into a dense neural network which performs classification. We do not mind losing spatial information here because we have already extracted the high-level features in a way that is invariant to spatial arrangements. To do classification, we usually use the activation function called **softmax**.

3.4 CNN applications

It turns out this architecture is also useful for many applications, such as object detection, segmentation, or probabilistic control.

Consider the object detection problem. The goal is to detect all objects of interest and output the labels with bounding boxes. This is difficult as we need to handle dynamic number of objects that may show up. One naive solution is to enumerate all possible sub-regions that objects of interest may show up then use the same CNN repeatedly on each region to identify the objects. This is obviously too computationally intensive.

So, there is another solution called **R-CNNs** which use human heuristics to identify regions that we think might have objects, then use CNNs to classify. Still, this is slow because we will usually need many regions for detection sensitivity. Since region proposals are manually programmed, the model is inevitably brittle. There has been some research that uses deep learning to learn the efficient region proposals.

Another solution to object identification problem is called **semantic segmentation**. Instead of identifying the objects directly, we use fully convolutional networks to output pixel-wise probability that the pixel belongs to each class. Then, we proceed on there with some decoding networks.

There are also further applications in the fields of medicine, self-driving cars, and much more.

4 Deep Generative Modeling

Ava P. Soleimany gave this fourth lecture on January 22nd, 2021. To make a point about facial recognition system, she started by polling us on whether we think the three displayed faces are of real people or generated with AI. It turned out all three were fake.

4.1 Learning approaches

In deep learning, there are two main types of learning approach: Supervised learning and unsupervised learning. In **supervised learning**, each data point x comes with a label y and we want to learn a functional mapping from x to y for general x. In **unsupervised learning**, we do not have the labels y at all. Instead, we let the machine come up with some insights about the data based on the parameters we give it. For example, it might try to find k clusters of data points that are, by some metric, similar.

Generative modeling takes training samples from some distribution as input and learns a model that represents that distribution. We can then use that model to generate new data. This is basically what a facial generation problem is! One issue with generative modeling is that of **biases**. Suppose we have images of human faces. It is possible that we end up with a model that is biased toward certain races or ethnicities because our data set has that inherent bias. There is a lot of work in the fields of **debiasing**.

We want to be able to also detect when we are dealing with new and unfamiliar input data. This allows our model to act accordingly, say, in self-driving cars where safety is very important.

4.2 Latent variables

The myth of the cave: Consider the prisoners who are forced to stay in a cave for a long time. They can only see objects from the outside world in the form of shadows cast upon their cave walls. In a sense, the shadows are their entire world and whatever names and descriptions they come up with are based on only their limited knowledge.

In generative modeling, we have this exact same problem: "Can we learn the true explanatory factors, e.g. latent variables, from only observed data?"

4.3 Autoencoders

Autoencoder is an **unsupervised approach** for learning a more compact feature representation from unlabeled training data (i.e. the representation has lower dimensionality). The way to do this

is to link two networks, encoder and decoder, together. The encoder tries to compress the input x into an encoding z, then the decoder tries to decodes z into $\hat{x}$. The goal is to make the constructed $\hat{x}$ as similar to the original x as possible. Namely, we minimize the loss $\mathcal{L}(x, \hat{x}) = \|x - \hat{x}\|^2$. Note that this is indeed an unsupervised learning approach because we do not need any labels at all! Obviously, if the **latent space**, the space of all possible encodings, has lower dimensionality, then this is a bottleneck that results in lower quality for the reconstructed data.

4.4 Variational Autoencoders (VAEs)

Instead of using a deterministic layer to output the encoding z, we use a stochastic approach. Specifically, we compute statistical aggregates such as the mean (μ) and standard deviation (σ) vectors to parametrize the latent space. The probability distribution dictated by those parameters are then used to generate z. We can denote the process of generation done by the encoder as $q_\phi(z|x)$ where ϕ is the probability distribution. Then, the decoder tries to compute $\hat{x}$ from z: $p_\theta(x|z)$. We want to minimize:

$$\mathcal{L}(\phi, \theta, x) = (\text{reconstruction loss}) + (\text{regularization term}).$$

The reconstruction loss is the same as in autoencoders. Meanwhile, the regularization term tries to limit the complexity of our learned probability distribution. We usually expressed this loss as:

$$D\left(q_\phi(z|x) \parallel p(z)\right).$$

The inferred latent distribution (what we are finding) is $q_\phi(z|x)$. The fixed prior on latent distribution (our domain knowledge) is $p(z)$.

A common choice of prior is a Normal Gaussian distribution:

$$p(z) \sim \mathcal{N}\left(\mu = 0, \sigma^2 = 1\right).$$

This encourages the encodings to distribute evenly around the center of the latent space and penalize the network when it tries to "cheat" by clustering points in specific regions (i.e. by memorizing the data). To calculate distribution difference, we usually use the **KL-divergence** metric, calculated as follows:

$$-\frac{1}{2}\sum_{j=0}^{k-1}\left(\sigma_j + \mu_j^2 - 1 - \log\sigma_j\right).$$

4.5 Intuition on regularization and the Normal prior

Let us think about why we would prefer a normal prior and similar distribution. First, we want **continuity**: We expect that points that are close in latent space generate similar content after decoding. Second, we want **completeness**: Sampling from the latent space should result in "meaningful" content after decoding. This means our output somewhat closely resembles our data set and is not completely random.

Note that encoding as a distribution without regularization does not guarantee these properties! The model would only learn to minimize the reconstruction loss, effectively mimicking the input data without generalizing well at all. With a normal prior, the latent space is clustered together around mean 0. This ensures continuity and completeness and enforces **information gradient** in the latent space.

4.6 Reparametrizing the sampling layer

If we use $z \sim \mathcal{N}(\mu, \sigma^2)$ directly, we cannot perform backpropagation easily. So, the idea is to write z as $z = \mu + \sigma \odot \varepsilon$ instead where μ and σ are the vectors and ε is a small random constant drawn from $\mathcal{N}(0, 1)$.

Another benefit of finding such latent variables allow us to perform **latent perturbation**: We can slowly increase or decrease a single latent variable to generate data with some changes. This gives meaning to each latent variable. For example, with face generation, a variable might represent face angle.

Ideally, we want all our latent variables to be independent from each other. **β-VAE loss** helps achieve this:

$$\mathcal{L}(\theta, \phi; x, z, \beta) = \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)] - \beta \cdot D_{KL}(q_\theta(z|x) \parallel p(z)).$$

Setting $\beta > 1$ constraints the latent bottleneck, which encourages efficient latent encoding. This is what we call **detanglement**.

Latent distributions allow us to create fair and representative data sets. By analyzing the latent variables, we can see what kind of biases our model might have. This is called **debiasing**.

4.7 Generative Adversarial Networks (GANs)

It is often difficult to learn from a complex distribution of data. So, there is an opposite idea where we try to create a **generator** network (G) which tries to create imitative data (X_{fake}) using small random noise (z) only. Then, there is a competing network, the **discriminator** (D), that tries to tell whether the data it gets is real or generated from the network. After iterations of training, the generator would model a probability distribution that is close enough to the real data that the discriminator cannot distinguish between real and fake data.

The discriminator tries to maximize the objective of distinguishing the fake data. Namely, we try to learn:

$$\underset{D}{\operatorname{argmax}}\, \mathbb{E}_{z,x}\left[\log \underbrace{D(G(z))}_{\text{fake}} + \underbrace{\log(1 - D(x))}_{\text{real}}\right].$$

Meanwhile, the generator tries to minimize being detected by the discriminator.

$$\underset{G}{\operatorname{argmin}}\, \mathbb{E}_{z,x}\left[\log \underbrace{D(G(z))}_{\text{fake}} + \underbrace{\log(1 - D(x))}_{\text{real}}\right].$$

GANs also work as distribution transformers. If we change our noise gradually, the output changes continually, giving us different output types.

4.8 Progressive growing of GANs

Researchers are studying GANs that are dynamically growing in size rather than having a fixed size. This allows for much more modeling power. **StyleGAN(2)** combines the idea of progressive growing with style transfer. This allows us to generate an image from a seed that mimics the style of some other source data.

Conditional GANs and **pix2pix** employ **paired translation** instead. The discriminator, D, classifies between fake and real **pairs**. The generator, G, tries to fool the discriminator. This paired translation is useful for changing one type of image to the other, for example, from a geographical map to a real street view.

CycleGAN learns transformations across domains with unpaired data. An example we are given is a rather impressive transformation from a video of a horse moving on green grass to a video of a zebra moving on yellow grass. Unlike usual GANs that try to create target data from Gaussian noise, CycleGANs try to generate target data from some source data in the other domain. In fact, CycleGAN is used to transform audio recordings from Amini's voice into Obama's voice in the first lecture! This is done by representing the audio as a spectrogram.

5 Deep Reinforcement Learning

The fifth lecture was given by Alexander Amini on January 22nd, 2021.

5.1 Reinforcement Learning (RL)

The field of reinforcement learning has actually existed before neural networks. However, deep learning has allowed it to develop much further. Amini found this field amazing because it is different from what we have learned in this class so far.

Until now, deep learning is constrained to the fixed data set we have. **Reinforcement learning** works on **dynamic environments** to learn to achieve a goal, often without human guidance. One can see that Reinforcement learning becomes very useful in the real world, especially in games and robotics. As an example, Amini showed a video of an experienced StarCraft player battling a trained artificial intelligence, AlphaStar. Even though Taylor said he would win 4–1 in the worst case, it turned out AlphaStar won against him 5–0.

Recall: there are three main types of learning problems. One is **supervised learning** where we want to learn a function to map from data to label. Another is **unsupervised learning** where we only have the data and want to learn the underlying structure (e.g. clustering to similar data). The third one we are learning about to day are the **reinforcement Learning** problems, where the data available is all possible state-action pairs. The goal is to maximize future rewards over many time steps.

5.2 Terminologies

Agent is something that can take actions in the environment. Our RL algorithm is the agent. **Environment** is simply the world in which the agent lives. The available **action** available or taken at time t is often denoted a_t. **Observations** are the reactions of the environment to our actions. Specifically, we get **state changes** denoted s_{t+1} and the immediate reward r_t. Since there can also be delayed reward, we define the concept of **total reward** which encapsulates the sum of rewards gained starting at time t into the future. The total reward is denoted R_t:

$$R_t = \sum_{i=t}^{\infty} r_i = r_t + r_{t+1} + \ldots + r_{t+n} + \ldots.$$

We usually consider the **discounted total reward (return)**. This metric places less importance on delayed rewards by adding a discount factor γ where $0 < \gamma < 1$. So, we have:

$$R_t = \sum_{i=t}^{\infty} \gamma^i r_i = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2}.$$

Q-function computes the **expected total future reward** that an agent in state s_t can receive by executing action a_t.

$$Q(s_t, a_t) = \mathbb{E}\left[R_t | s_t, a_t\right].$$

If we are given an oracle that returns the value of this function for any arguments, then we can infer the best course of action to take at each state s.

A strategy detailing which action to take at each state s is called a **policy**, denoted $\pi(s)$. The best policy $\pi^*(s)$ simply chooses an action that maximizes the expected total future reward:

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}\, Q(s, a).$$

5.3 Reinforcement Learning Algorithms

There are two types of algorithms for reinforcement learning we will focus on today. First, we learn about the **value learning algorithms** which try to approximate Q and infer the best policy. Then, we discuss **policy learning algorithms** which directly finds the best policy.

Consider the game Atari Breakout. The agent in this game is the paddle at the bottom of the screen. The available actions are left and right movements. The state is the ball's position and velocity and the remaining blocks on the top of the screen. The goal is to hit as many blocks as possible by moving the paddle without dropping the ball.

It can be very difficult for humans to accurately estimate Q-values. Consider two seemingly equivalent cases: (1) The ball is right in the middle of the paddle and we choose not to move. (2) We are moving the paddle to hit the ball just right on the edge. While the first strategy is much more intuitive for humans, it turns out the second strategy completes the game faster.

5.4 Deep Q Networks (DQN)

We could use a simple deep NN to map from all state-action pairs to the Q-function values. However, it is much more convenient for the network to only take the state as the input and output the rewards for all actions available at this state.

When training a DQN, we do not have any data to work with. So, we must try actions and update our knowledge of the Q function.

For each state s, we evaluate for all possible actions a what the best case scenario reward is. If r is the immediate reward from taking action a, then the best total reward is

$$r + \gamma \max_{a'} Q\left(s', a'\right).$$

We can set our neural network to predict $Q(s, a)$ to be as close to this target value as possible. A version of mean squared loss dubbed **Q-Loss** here works great:

$$\mathcal{L} = \mathbb{E}\left[\|r + \gamma \max_{a'} Q\left(s', a'\right) - Q(s, a_)\|^2\right].$$

Note that this method also generalizes to other Atari games. If we take the game picture as the input, we can use convolutional neural networks from the previous lecture to first comprehend the game, and then use fully-connected neural networks to determine the Q function using Q-Learning. This method has allowed AIs to surpass human-level skill in the majority of Atari games.

Note the downsides of value learning algorithms like this. This network can only model situations with small and discrete action spaces. It cannot deal with continuous action spaces. It is also not flexible: Q function calculation is completely deterministic, so it cannot learn stochastic policies.

5.5 Policy Learning Algorithms

Instead of approximating Q-function, we directly optimize the policy $\pi(s)$ to target the optimal one. We represent our policy as a probability distribution: At state s, how likely should we take action a_i? The probability is denoted $P(a_i|s)$. Note that $\sum_a P(a|s) = 1$.

We can use this to deal with continuous action spaces. Instead of outputting the probability for each discrete action, we only output the parameters representing a probability distribution, say, a Gaussian curve.

5.6 Self-driving cars

The agent is the vehicle. The state is all the input sources we have (e.g. camera). The action is the steering wheel angles to choose from. The reward is total distance traveled.

To create self-driving cars, we first initialize the agent (e.g. randomly) and run the policy until termination. Record all states, actions, and rewards during that run. Use those state-action-reward tuples to modify the weights. Repeat until convergence.

Since we want our weight updates in each iteration to incentivize "good" actions and discourage "bad" actions, we need to design a good loss function. It turns out the negative log-likelihood loss function works:

$$\mathcal{L} = -\log P\left(a_t|s_t\right) \cdot R_t.$$

The problem with the "run policy until termination" step is that it is not practical in the real world, especially not with driving cars. So, we need to resort to simulation engines. Researchers try to come up with the most accurate engines possible as that allow the model to be deployable in the real world.

5.7 The Game of Go

The game of Go is interesting because the number of possible positions grow exponentially with very high base term. Reinforcement learning successfully outperformed human professionals.

Google's AlphaGo, premiered in 2016, used a supervised learning policy network to first learn human expert positions. Then, through self-play, it learned how to play better than humans.

AlphaGo Zero was developed two years later. It beat AlphaGo and human players without entirely through self-play training (without human data). AlphaZero generalized the model to beat any games as long as it knew the rules. Finally, MuZero managed to learn to play many games without even knowing the game rules in the beginning. The key idea is to take into account tree searching in the Q function/policy probability calculation.