APS360 Applied Fundamentals of Deep Learning

These notes were taken from lecture recordings from a past year. They were done fast and are low-quality. Please use the other (more recent) uploaded notes instead.

Week 2

Neurons

- Each neuron has n inputs x_i and produces an output $y = f\left(\sum_i w_i x_i + b\right) = f(\boldsymbol{w} \cdot \boldsymbol{x} + b)$
  - w_i are the weights, one for each input
  - b is the bias (a weight with no input)
  - f is the activation function that determines the output
    - When f is linear, this becomes a support vector machine
    - b is related to the offset of the hyperplane that cuts the input space in half
- Linear activation functions are not a good idea because they do not benefit from multiple layers
  - The composite of multiple linear operators is also a linear operator, so composing multiple layers does not give you anything more
  - Problems are usually nonlinear so using only linear activation functions limits what you can do
- Some activation functions:
  - Perceptron: binary activation function, $f(x) = \text{sign}(x)$ or unit step
    - The point at which the output changes is the *decision boundary*
    - Not differentiable, continuous, or smooth, presents a problem for optimisation
    - Used in the early days
  - Sigmoid: output in $[0, 1]$, $f(x) = \tanh(x)$ or $f(x) = \dfrac{1}{1 + e^{-x}}$ etc
    - Easily differentiable, smooth, and continuous
    - Large input saturate the neuron (vanishing gradient problem)
    - Still used but most common before 2012
  - Rectified linear unit (ReLU): linear when positive, 0 when negative, $f(x) = \max(x, 0)$
    - At $x = 0$ the gradient is defined to be 0
    - Leaky ReLU has a small slope in the negative region instead of 0, which can be parametrized in PReLU (common default is 0.1)
    - Derivatives are very easy to compute
  - Smooth ReLU: $f(x) = \dfrac{x}{1 + e^{-x}}$ (SiLU) or $\dfrac{1}{\beta} \log\left(1 + e^{\beta x}\right)$ (SoftPlus)
    - Work better or on par compared to regular ReLU
    - More computation for derivatives
- Networks can have several layers, with the final layer being the output layer and previous layers being hidden layers
  - In a fully-connected network each neuron receives inputs from each neuron in the previous layer
  - In a feed-forward network information only flows forward from one layer to a later layer
- Having more hidden layers lets you get better approximations of functions
  - A theoretical network with infinite layers is a universal function approximator
  - With more layers, gradient computation requires backpropagation, which allows the distribution of errors to neurons not adjacent to the output layer
- Each layer can be viewed as learning more and more abstract features
  - Previous layers transform the raw data into a higher-level form that allows more abstract features to be picked out
  - Eventually the raw input is transformed into a linearly separable form

Loss

- To train a neuron, we compute a prediction and feed it into the loss function along with the ground truth, and then adjust the weights (and biases) to minimize the loss
  - Forward passes through the network are used for both normal inference and training
  - Backward passes are used for training only
- Small loss indicates that the network's prediction matches the ground truth
- Loss will initially decrease with training, but it may start increasing again at some point
  - For this reason the dataset is usually split into training, validation, and testing
  - Training is stopped when loss on validation starts increasing
  - The testing dataset is used to evaluate the network's performance on unseen data (validation is not used in training, but it influenced when training was stopped)
- For classification problems, the raw output (*logits*) are normalized into a final probabilistic output
  - A softmax function can be used: $\text{Softmax}(x) = \frac{e^{x_i}}{\sum_{k=1}^{K} e^{x_k}}$
- Some loss functions:
  - Mean squared error (MSE): mostly used for regression, $\frac{1}{N}\sum_{n=1}^{N}(y_n - t_n)^2$
  - Cross entropy (CE): also used for classification, $-\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} t_{n,k}\log(y_n, k)$
    - n are the training samples and k are the number of classes
    - More suitable for classification since MSE does not handle probabilities correctly
  - Binary cross entropy (BCE): for binary classification, $-\frac{1}{N}\sum_{n=1}^{N}(t_n \log(y_n) + (1 - t_n)\log(1 - y_n))$

Gradient Descent

- Each layer can be represented as $\boldsymbol{y} = f(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$
- Training the network involves optimizing the weights, so we wish to find $\frac{\partial E}{\partial w_{ji}}$
- Once the gradients are found, we take a step in the direction of the gradient
  - $w_{ji}^{t+1} = w_{ji}^{t} - \gamma\frac{\partial E}{\partial w_{ji}}$ where γ is the learning rate
  - γ requires fine tuning; too big and the network bounces around the minimum, too small and it gets stuck in local minima and is too slow to learn
- In practice we need to keep a set of remembered points in the optimization since we might overshoot the minimum after reaching it
- Example: For a single layer with MSE error and sigmoid activation function, compute $\frac{\partial E}{\partial w_p}$
  - Let $y = f(a), a = \sum_p w_p x_p + b$
  - $\frac{\partial E}{\partial w_p} = \frac{\partial E}{\partial y}\frac{\mathrm{d}y}{\mathrm{d}a}\frac{\partial a}{\partial w_p}$
  - $\frac{\partial E}{\partial y} = \frac{\mathrm{d}}{\mathrm{d}y}(y - t)^2 = 2(y - t)$
  - $\frac{\mathrm{d}y}{\mathrm{d}a} = \frac{\mathrm{d}}{\mathrm{d}a}\frac{1}{1 + e^{-a}} = (1 - y)y$
  - $\frac{\partial a}{\partial w_p} = x_p$
  - $\frac{\partial E}{\partial w_p} = 2x_p(y - t)(1 - y)y$

Week 3

Hyperparameters

- Parameters that are not optimized during learning (i.e. not the weights), e.g.:
  - Batch size
  - Number and size of layers
  - Activation function
  - Learning rate
- Tuning is expensive since each set of hyperparameters requires the entire network to be retrained
  - Since the weights are seeded randomly in practice we need to train multiple times for the same set of hyperparameters
- Could take a smaller subset of training data to find optimum for hyperparameters
- Hyperparameter turning could be done through a uniformly distributed grid search or with a random search (only applies for continuous parameters)

Gradient Descent Optimizers

- The learning problem turns into an optimization problem when a loss function is assigned
- The *credit assignment problem* is the problem of determining how and how much each parameter affects the performance
- Gradient descent is one such optimizer
- Stochastic gradient descent (SGD) takes a subset of the data in each iteration and takes a step of gradient descent based on this data only (unlike gradient descent which uses the entire dataset)
  - Since we don't use the entire data set, each iteration takes less time (but may not be faster overall)
  - Optimization path is more erratic (less direct), but will often result in a better set of weights since the search is more global
- Mini-batch gradient descent will compute average loss for n samples, and then take a gradient descent step to optimized the average loss of the batch
  - *Epochs* are the number of times all the training data is used
  - *Iterations* are the number of steps taken
  - e.g. 1000 samples with a batch size of 10 gives 100 iterations per epoch
  - Typically 20-30 is a good number of epochs
- *Ravines* are areas where the gradient in one dimension is much steeper than in the other; gradient descent will jump around in the steep direction and move slowly in the shallow direction
  - We can remedy this by adding a *momentum* term
  - $v_{ji}^t = \lambda v_{ji}^{t-1} - \gamma \dfrac{\partial E}{\partial w_{ji}}$
  - $w_{ji}^{t+1} = w_{ji}^t + v_{ji}^t$
  - $\lambda = 1 - \gamma$
- In *adaptive moment estimation* (Adam) each weight has its own learning rate; this incorporates both momentum and adaptive learning rate
- Learning rates typically depend on the problem, optimizer, batch size (larger batch requires larger learning rates), and training stage (rate should be reduced as training progresses)

Normalization

- Inputs to the network are normalized as $\hat{X}_i = \dfrac{X_i - \mu_i}{\sigma_i}$
  - This standardizes the input data so the model doesn't pay more attention to features with larger range
  - This normalizes for the first layer only
- For subsequent layers, each layer's activations are normalized based on mean and variance for the entire mini-batch (batch normalization)
  - Batch normalization allows a higher learning rate, regularizes the model and makes it less sensitive to initialization

  - However this cannot work with SGD and has no effect with small batches
- Alternatively normalization is applied at the output of each layer before it reaches the next layer

Regularization

- Prevents overfitting
- *Dropout*: randomly setting weights to zero during training (dropping activations) with probability p, and during inference multiply weights by $1 - p$ to keep the same distribution
  - This encourages more robust models
- *Weight decay* (aka *L2 norm regularization*): adding the L2 norm of the weight vector to the loss function
  - This reduces each weight multiplicatively in each iteration
  - Effectively changes the update equation to $W_{t+1} = W_t - \gamma \left(\alpha W_t + \frac{\partial E}{\partial W} \right)$
- *Early stopping*: stopping the training when loss starts to increase
  - With "patience", start a counter when the loss starts to increase and reset it when the loss decrease; if it reaches a certain point then stop training

Confusion Matrix

- True positive/negative: when prediction and true label agree
- False positive: when prediction is positive but label is negative
- False negative: when prediction is negative but label is positive
- Common metrics:
  - *Accuracy*: probability of output being correct (sum of true positives and negatives over all samples)
  - *Precision*: probability of true positive, given positive prediction (true positives over sum of true and false positives)
  - *Recall*: probability of true positive, given positive label (true positives over sum of true positives and false negatives)
  - *F1 score*: $2 \times \frac{\text{recall} \times \text{precision}}{\text{recall} \times \text{precision}}$

Week 4

Convolutional Neural Networks (CNNs)

- Simply feeding image pixels directly to the input layer requires large networks and does not take into account image geometry well
- In a CNN, a convolutional filter is first used on the image
  - The kernel is to be learned by the network
    - Note the kernel may include a bias term
    - Also initialized randomly
  - This will detect low-level localized features first globally across the image, which is then fed to hidden layers
  - This is opposed to an MLP (multi-layer perceptron) network (i.e. conventional)
- The CNN Is split into the encoder (feature detection) and classifier stages
  - The feature detection stage consists of (multiple stages of) convolutions + ReLU and pooling
    - Features get more and more abstract and higher level with each convolution
  - The final result is flattened and fed to the classification stage which operates as usual
- CNN design parameters:
  - *Padding*: the edges of the image could be padded with zeros so the output of the convolution stays the same size, so information around the edges is not lost
  - *Stride*: the distance between consecutive positions of the kernel (allows control over the output resolution)
- Each dimension has an output size of $o = \left\lfloor \frac{i + 2p - k}{s} \right\rfloor + 1$ where i is the image dimension, k is the kernel dimension, p is the amount of padding (each side) and s is the stride

- For colour images, the kernel becomes a 3-dimensional tensor, operating on all 3 channels at the same time; the image would be $3 \times i \times i$ and kernel $3 \times k \times k$
  - This is like applying a separate kernel to each channel and then summing the results for each pixel
- To detect many different features, we can have multiple kernels (filter depth)
  - The number of kernels is the number of output channels
- e.g. colour input image of $3 \times 28 \times 28$ using kernels $5 \times 3 \times 8 \times 8$ has 3 input channels, 5 output channels and $5 \times 3 \times 8 \times 8 + 5$ trainable weights (including biases)
- As we go through the convolutional layers, the filter depth increases and feature map decreases in size
  - The features we pick out are getting lower and lower in resolution but there are more features
- Generally, networks with more depth do better, but very deep models may be untrainable due to vanishing or exploding gradients
  - Improvements in the past decade such as improved weight initialization for ReLU, normalization, and residual connections (discussed later) address this
  - For very deep networks we need few fully connected layers for classification, sometimes even just one, since the features are already very good

Pooling

- We often want to reduce the size of the output from the convolutional layers before we pass it to the ANN
- Pooling is essentially another convolution over the output, but the kernel is not tunable:
  - *Max pooling*: taking the max value in the entire area that the kernel covers
  - *Average pooling*: taking the average of all the values in the area the kernel covers
- Or use strided convolutions, i.e. another convolutional layer with bigger strides
  - This is more powerful since the weights can be learned, but pooling is a fixed operation
  - However it introduces more parameters to be learned and makes inference also more expensive

Week 5

Visualizing CNNs

- *Saliency map*: heatmap highlighting the areas of the image that are relevant for the classification
  - For a particular input image, the gradients are computed and the max absolute value is taken across all channels and visualized
  - This gives visual intuition of what the network is focusing on
  - Not too useful in general

Modern CNN Techniques

- *Augmentation*: Applying class-preserving transformations to the input, i.e. modify it slightly in such a way that the label is not changed, to generate additional training data
  - Helps makes network more robust and general
  - e.g. cropping, resizing, rotation, color distortion, blurring, adding noise, cutout, Sobel filter (edge highlight), etc
- *Pointwise convolutions*: Applying pixel-wise, possibly nonlinear transformations that map each pixel into a higher or lower dimensional space, e.g. turning RGB pixels into a single value
  - Used in most modern CNN architectures
- *Auxiliary loss*: adding additional loss functions partway through the network, and optimizing the total loss
  - i.e. adding intermediate classifiers and making the final loss the combination of the intermediate and final losses
    - Note these intermediate classifiers also need their own fully connected layers; i.e. we're taking the intermediate output of the convolutional layers and using fully connected layers to classify them with the same labels, so the intermediate classifiers should have the same labels
  - This helps with the vanishing and exploding gradient problem and can also help with overfitting

- *Inception block*: using a mixture of filter sizes on one layer
  - Normally we need to fit all these into a tensor so they need the same dimensions, but we can break it up
  - The most important features can be mostly learned with just 3×3 filters, with a few larger ones added
  - Improves parameter efficiency
- *Stacked convolutions*: simple architecture made of simple stacked blocks
  - VGG (Visual Geometry Group, Oxford) showed that stacked 3×3 filters can approximate larger filters more efficiently
  - Stacked filters apply the same filter (i.e. with the same weights) multiple times to approximate a larger sized filter with only 3×3 filters
- *Residual networks* (ResNets): using skip connections to provide deeper layers more direct access to outputs of earlier layers
  - The input to a deep layer comes from the layer directly before it, added with the output of a layer several levels before, skipping intermediate layers
  - This helps with vanishing gradients and allows training very deep networks

Transfer Learning with Embeddings

- The output produced after all the convolutional layers but before the classification layers is an *embedding*
  - This is a set of features that contain everything needed to classify an image
  - These embeddings can be reused to train new networks
- After training on large datasets, the convolutional layers learn something general about representing images that is useful across a range of tasks, so we can reuse them for different tasks
- To transfer learning to a new problem, the classification (fully connected) layers are removed, and the weights in the convolutional layers are frozen; then new layers are added that are more suitable for the new task and retrained
  - This lets us use very powerful pre-trained networks for customized tasks
- Since the weights in the CNN are frozen, they won't be trained and the CNN is used as a feature extractor
  - We can alternatively also train these weights but with very small learning rates, which is known as *fine-tuning*
  - This helps the CNN adapt to the new task

Week 6

Unsupervised Learning (Overview)

- So far we've only discussed *supervised learning*, which requires data with labels, or ground truth
  - This is expensive to generate since the labels need to be manually created, and we need a very large amount of it
  - Ground truths can also have noise, e.g. humans may make errors when labelling data
- In *unsupervised learning*, models instead learn to recognize patterns without explicit supervisory signals
  - e.g. clustering, probability density estimation, dimensionality reduction
  - *Semi-supervised learning*: a small subset of the data is labelled, but network is mostly trained on unlabelled data
  - *Self-supervised learning*: supervision/labels are automatically created
    - e.g. cutting out a part of an image and predicting what was there

Autoencoders

- An *autoencoder* finds efficient representations of input data by having two components that could reconstruct the input:
  - Encoder: converting inputs to an internal representation
    - Often a form of dimensionality reduction

        * e.g. converting the MNIST dataset images into a latent space of just 2 dimensions
    - Decoder: converting the internal representation back to a form matching the input data
        * This is a generative network
- The combination of the encoder and decoder makes the full autoencoder, which has the same dimension of output as the input
    - The output of the encoder and input of the decoder is the *bottleneck layer*, which is an (often) lower dimensional representation
    - Due to the dimensionality reduction, the autoencoder is forced to learn only the most important features in the input data and drop the unimportant ones
- Autoencoders generally have an hourglass shape, which the bottleneck layer in the middle; they are often symmetric but does not have to be the case
- We want the output to approximately match the output, so the loss will simply compare the input and output, with no need for labels
- Applications:
    - Feature extraction and dimensionality reduction
    - Unsupervised pre-training: training the encoder, and attaching an ANN to it for a classification problem
    - Generating new data: sampling from the latent space and pushing it through the decoder to generate new input
        * We can compute embeddings of two images, interpolate between them, and then pass it through the decoder, which will result in an image somewhere in between the two
        * If we sample purely randomly from the latent space, we might generate something that doesn't make sense, since the latent space can be disjoint and non-continuous
        * Variational autoencoders address this issue
    - Anomaly detection/noise reduction: the autoencoder is bad at replicating outliers (since they are not learned), so these will be filtered out
        * Conversely noise can be artificially added to the input data to force the autoencoder to only learn important features
- To assess the autoencoder performance we can simply visually compare the input and output; however, perfect reconstruction can be a case of overfitting

Variational Autoencoders (VAEs)

- Used for generative purposes and addresses the continuity issues discussed above, by imposing a constraint on the latent space to make it smooth
    - Can be thought of as an autoencoder that is trained so that the latent space is regular enough for data generation
    - This avoids the issue of meaningless data being generated from randomly sampling the latent space
- They are probabilistic – outputs are non-deterministic even after training
- Instead of a fixed embedding the encoder generates a normal distribution with some mean and standard deviation, from which the embedding is randomly sampled; the decoder then takes the embedding sampled from the distribution given by the encoder and tries to reconstruct the input
    - Mathematically the encoder provides a prior distribution $p(z|x)$ for embeddings z conditioned in input x; then embeddings are sampled from this distribution and reconstructed by the decoder
- We want the latent space to be *regular*: continuous (points that are close should generate outputs that are similar) and complete (points should not generate meaningless data)
    - The model can overfit and reduce to a simple autoencoder in 2 ways, either by giving very low variances, or having very different means; both will lead to an irregular latent space
    - Therefore we want to force the priors generated by the encoder to have a certain variance and have means that are close together
    - To do this, we add a regularization term in the loss function that compares the prior against a standard normal distribution
        * Use *Kullback-Leibler (KL) divergence*: $D_{KL}(P \parallel Q) = \sum_{x \in \mathcal{X}} p(x) \log \left(\frac{p(x)}{q(x)} \right)$

    * For a multivariate Gaussian and standard normal: $\frac{1}{2}\sum_{i=1}^{N}\left(\mu_i^2+\sigma_i^2-(1+\log(\sigma_i^2))\right)$
- The total loss is the sum of the reconstruction loss and the regularization term
  - These are two conflicting goals that together prevent overfitting
- The variances in the output of the encoder give us bounds for sampling the latent space, so that our generated results will look a certain way (e.g. a certain digit instead of a merge of two digits)

Convolutional Autoencoders

- For convolutional networks, we now have the problem of going from the embedding back to the image and undoing our convolutions
  - This is the problem of upsampling
  - We could simply not use convolutions and just have ANN layers, but this has the same downsides as an ANN vs CNN
- *Transposed convolutions* are the inverse of convolutions and can map 1 pixel to $k \times k$ pixels
  - For each pixel, the entire kernel is multiplied by the pixel value and added to the output image; when outputs overlap they are summed
    * Similar to using a stamp
- The output dimension is given by $o = s(i-1) + (k-1) - 2p + p_o + 1$ where p_o is the output padding
  - Padding works in the opposite way; since the output of transposed convolution is larger than the input, a positive padding will chop off the edges of the output and reduce its size
  - The output size could be ambiguous for $s > 1$, so the output padding resolves this by effectively increasing the output shape on one side
    * e.g. for a normal convolution, both 7×7 and 8×8 gives a 3×3 output for $k = 3, s = 2$; when going backwards, output padding allows us to determine which output size to pick
    * Used to determine output shape only (doesn't actually pad zeros to the output)
  - This allows us to get the exact same size back by applying a convolution and then a transposed convolution

Pre-Training with Autoencoders

- We can first train the autoencoder on unlabelled data, then take only the encoder, attach an ANN, and use it to train a classification problem
- The encoder portion is used as a feature detector like in the case of transfer learning with CNNs
- During the supervised classification problem training the weights in the encoder are further fine-tuned
  - After this, it can be reinserted back into the autoencoder for better performance

Self-Supervised Learning

- *Proxy-supervised tasks* are tasks such that the labels can be generated automatically for free and solving the task requires the model to "understand" the content
  - We want to devise the tasks such that the model is forced to learn robust representations
  - e.g. rotating an image and having the network guess how much the image was rotated from the original (RotNet)
    * For this task, the network needs to learn the concepts of the objects in the images to see that they have been rotated
- In *contrastive learning* we have pairs of samples that are fed to the network, and the loss is computed in latent space, with the embeddings expected to be equal
  - We train the network so that "similar" input results in similar embeddings and different inputs result in embeddings that are far apart

Week 7

Processing Words

- Input words to a model can be encoded with one-hot encoding, with one dimension for each possible word
  - This leads to very large input vectors but is better than a one-dimensional encoding
  - Using one dimension and assigning a number to each word would not work well because the number assignments are arbitrary, and it bottlenecks the network to a 1 dimensional representation
- Given a word we want to find its embedding in latent space, such that similar words are close together
  - Important to note that the meaning of a word comes from its context
  - word2vec and GloVe are two commonly used models
- word2vec uses two architectures: skipgram and CBOW (continuous bag of words)
  - Skipgram takes a word and tries to predict its surrounding context, e.g. given the centre word, find 2 words that could appear before it and 2 words that could appear after
    * Training data is drawn from lots of existing text
    * The one-hot encoded input word vector passes through hidden layers and goes to an output layer
    * Output is softmax, essentially giving a probability to find each word
    * Advantages:
      - Works well with smaller datasets
      - Better semantic relationships (e.g. relating "cat" and "dog")
      - Better representation of words that appear less frequently
  - CBOW does the opposite and predicts the centre word from context
    * Advantages:
      - Trains faster since the task is simpler
      - Better syntactic relationships (e.g. relating "cat" and "cats")
      - Better representation of words that appear more frequently
  - In both cases the output layer is used only for training; we discard it after because we only care about the internal embedding
- GloVe is another common model that encodes global information into embeddings
  - GloVe computes co-occurrence frequency counts for each words; the number of times word i appears in the context of word j is stored in a matrix as X_{ij}
- To compare similarity of word embeddings, we have 2 common choices of distance measure:
  - Euclidean distance: L2 norm of difference of embeddings
    * $\sqrt{\sum_{i=1}^{d}(x_i - y_i)^2}$
  - Cosine similarity: cosine of the angle between embeddings (equivalent to dot product of the embeddings divided by product of their magnitudes)
    * $\frac{\sum_{i=1}^{d} x_i y_i}{\sqrt{\sum_{i=1}^{d} x_i^2}\sqrt{\sum_{i=1}^{d} y_i^2}}$

Recurrent Neural Networks (RNNs)

- Processing text is difficult because the input size is variable
- We need an architecture that allows us to remember previous information
- In an RNN layer, the output is determined by the input in addition to a hidden state (i.e. a memory)
  - This hidden state is updated each time a forward pass is performed
  - Previous input data can affect later output through the hidden state
- Words are passed in one at a time to the RNN
  - Hidden state tracks the context
  - Output from the final word is the final prediction, which is affected by all previous word inputs
- RNNs often suffer from *memory loss*, i.e. the inputs in earlier stages can be forgotten

- RNN update equations:
  - $h_t = \sigma_h(W_h x_t + U_h h_{t-1} + b_h)$
  - $y_t = \sigma_y(W_y h_t + b_y)$
  - x_t is the input, y_t is the output and h_t is the hidden state at time t
  - σ_h, σ_y are activation functions for the hidden state and output (sigmoid is used often)
  - Notice that the hidden state is updated before the output is computed, so the output is computed based on the current hidden state
- If we concatenate the input and hidden state, and concatenate the output and hidden state, and make these our new input/output, then this behaves simply as a fully-connected NN

Week 8

Vanilla RNNs – Limitations

- RNNs face two major problems when working with long sequences: memory loss and vanishing/exploding gradients
- Gradients tend to vanish or explode since the hidden state is multiplied by a matrix each time
  - If the matrix has eigenvalues greater than 1 in absolute value, then with very long inputs the gradient explodes, otherwise it vanishes
  - Gradient clipping (i.e. constraining the gradient to within a certain threshold) can address exploding gradients
  - Skip connections can address vanishing gradients and memory loss
    - Ideally we want skip connections to all previous states, but this is too expensive
    - These need to be carefully chosen

Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) Architectures

- LSTM consists of both a long-term memory (*cell state* C_t) and short-term memory (context or hidden state h_t)
- 3 gates are used to update memory:
  - Forget gate (long-term): past short-term memory contributing to long-term memory
  - Input gate (long-term): current input contributing to long-term memory
  - Output gate (short-term): updated long-term memory contributing to the new short-term memory
- In LSTM, the past short-term memory and current input are first used to update the long-term memory, then new short-term memory is derived from the new long-term memory, and output comes from the short-term memory
- GRUs combine the forget and input gates into an update gate, and cell state and hidden state are merged
  - The short and long-term memory are combined into one
  - This is more efficient than LSTM while having a similar performance
- GRUs and LSTMs can handle longer sequences better and are generally easier to train with better performance

Deep & Bidirectional RNNs

- A typical state in an RNN relies on only the past and present
  - For some applications (e.g. translation), the prediction can also depend on the future
- A bidirectional RNN uses 2 unidirectional RNN layers in opposite directions
  - One of the layers will receive the input tokens in forward order while the other in reverse order
  - The overall output is the result of concatenating the corresponding outputs of the two layers
- RNNs can also be stacked to form deep RNN architectures
  - As with CNNs, this is useful for more abstract representations
  - The first layers are better for syntactic (grammar) tasks
  - Later layers are better for semantic (meaning) tasks

Sequence-to-Sequence Models

- So far we have focused on many-to-one tasks and many-to-many tasks where the output is the same length
  - Other tasks might require slightly different RNN setups, e.g. translation, image captioning, etc
  - There are also one-to-one, one-to-many, and many-to-many tasks with different output lengths
- To generate sequences we again use an autoencoder
  - The encoder processes tokens one at a time, and the hidden state represents context of all tokens read so far
  - The decoder generates tokens one at a time, and the hidden state represents all the tokens to be generated
- We also need to know when to stop a generated sequence
  - For this we use dedicated control symbols to denote beginning of sequence and end of sequence (BOS/EOS)
- During training, the RNN is trained to generate one particular sequence in the training set at a time
  - BOS is fed to the model and we compare the output with the first word that we want (cross entropy)
  - Then we feed in the first expected word (i.e. the ground truth label) and compare output to the expected second word and so on, until EOS
    * Feeding in the ground truth label to get the next word instead of the previously generated output is known as *teacher forcing* and helps the model train faster
- During inference, we don't want to always select the highest probability output since we want diversity, but we cannot have too much diversity or else there would be grammatical errors
  - We sample from the predicted distributions using one of 2 strategies:
    * Greedy search: simply select the token with the highest probability
      - Maximize $p(t_1)p(t_2)\cdots p(t_n)$
    * Beam search: search for a sequence of tokens with the highest probability within a window
      - Maximize $p(t_1)p(t_2|t_1)\cdots p(t_n|t_{n-1},\cdots,t_1)$
  - Softmax temperature scaling: scale the input logits to the softmax by a temperature τ
    * $\frac{e^{z_i/\tau}}{\sum e^{z_i/\tau}}$
    * A higher temperature spreads out the distribution more
    * At lower temperature, the logits are larger and the model is more confident
      - This results in higher quality samples but less variety
    * At higher temperature, the logits are smaller and the model is less confident
      - This results in lower quality samples with more variety

Week 9

Generative Adversarial Networks (GANs)

- Autoencoders are not great at tasks such as generating faces
  - Autoencoders try to eliminate noise, so they apply a kind of averaging
  - Therefore the output is often blurry
- In a GAN there are 2 competing models: a *generative model* and a *discriminative model*
  - The generative model takes an embedding and tries to generate an output matching something in the dataset
    * This is an unsupervised task
    * The output is a binary true/false so the output layer is always a single neuron
    * Generative models can be *unconditional* or *conditional*
      - Unconditional models take random noise or a fixed token as input; there is no control over what category they generate
      - Conditional models take as input a one-hot encoding of the target category and random noise, or an embedding from another model (e.g. CNN); we have control over the category to be generated

  - The discriminative model takes some input and determines whether it is real (from the training dataset) or fake
    * This is a supervised task since the output is a real/fake label
    * Fed with either a real sample from the training dataset or a fake sample generated by the generative model
- The generator tries to fool the discriminator by generating real-looking images while the discriminator tries to distinguish between real and fake images from the generator
  - The discriminator loss is simply binary cross entropy since we know if an input is real or fake
  - The generator loss is defined based on the discriminator output; we want the discriminator to output real every time (use BCE)
    * To compute the gradient for the generator we have to backpropagate through the discriminator as well
    * Only the generator's weights are updated in this step since we don't want to make the discriminator worse
- To generate a specific class of output from the generator, we use a conditional model, where a label is also passed in along with random noise
  - The same label is also passed to the discriminator during training
  - e.g. one-hot encoding for a digit from the MNIST dataset, or an image in greyscale for a model that converts greyscale to colour

Training GANs

- To train, alternative between training the discriminator and generator
  - First train the discriminator for k steps: sample m noise samples and m real samples, pass through the generator and then discriminator, and compute gradients with BCE
  - Then train the generator: sample m noise samples, pass through generator and discriminator, compute gradients with BCE (with "ground truth" being a real label from the discriminator)
- Since the generator loss uses the discriminator, if the discriminator is too good then small changes in the generator weights won't change the discriminator output
  - This leads to a vanishing gradient problem
- If the discriminator gets trapped in a local optimum, it cannot adapt to the generator so the generator can fool it by only generating one type of data
  - This leads to *mode collapse* where the generator only generates one class of data (e.g. only a single type of digit in the MNIST dataset)
  - We don't want this since we want diversity in the generator output
- Since there are 2 competing processes, GANs are very hard to train and take very long
  - Plotting the training curve doesn't help much
  - Difficult to see whether there is progress – training curve doesn't help much
- To speed up we can use leaky ReLU, batch normalization, and regularizing the discriminator weights and adding noise to discriminator inputs

Adversarial Attacks

- Goal: choose a small perturbation ϵ on an image x such that a neural network f misclassifies $x + \epsilon$
- To do this we optimize ϵ as a parameter; we want to minimize the probability that the network classifies the image with noise added as the correct class
  - This can be targeted or non-targeted
  - For non-targeted attacks we minimize the probability that $x + \epsilon$ is classified correctly
  - For targeted attacks we maximize the probability that $x + \epsilon$ is a certain target class
- In a *white-box attack* we know the model already, so we can use the architectures and weights to optimize ϵ
- In *black-box attacks* we do not know the architectures and weights of the network
  - A substitute model mimicking the target model can be used
  - Adversarial attacks often transfer across models if they are using the same dataset

- Defence against adversarial attacks is an active area of research; failed defences include adding noise at training time or test time, averaging models, weight decay, dropout, or adding adversarial noise at training time

Week 10

Attention Mechanism

- The main building blocks of transformers are the *attention mechanism*
  - Intuitively, when humans look at something, we focus a lot on certain regions and pay less attention to surrounding regions
  - The network learns an *attention score*, which indicates the importance of different parts of the data
  - The attention score can then be used to aggregate the input
- Example: pooling for a system classifying tweets
  - We can take every word in the sentence, find its embedding (using word2vec/GloVe) and take the sum or average to get an aggregate result that can be passed to a classifier network
    - However, this assumes all words have the same importance and order information is lost
  - We can use a fully connected network that takes word embeddings and gives a score for each embedding, then normalize the scores (softmax), then use the attention scores to take a weighted sum of the word embeddings
    - This network is trained end-to-end with the classifier
    - This solves the problem of word weighting, but not order
- 2 types of attention:
  - *Cross-attention*: between two sequences, e.g. in translation
  - *Self-attention* (or *intra-attention*): input with respect to itself, e.g. for a token, compute the attention for all other tokens in the same sequence
- Given two embeddings we have many different methods to compute their attention scores, e.g. dot product, cosine similarity, bilinear, fully-connected network

Transformer Networks

- RNNs are sequential in nature; the iterative nature means we cannot take full advantage of vectorization or GPUs for training
- *Transformer networks* are based on self-attention modelled as a neural dictionary: it retrieves a value v_i for a query q based on a key k_i
  - Values, queries, and keys are d-dimensional embeddings
  - Rather than return a fixed value for a query, it uses a soft retrieval: retrieving all values and then computing their importance with respect to the query based on the similarity between the query and their keys
    - The result is a kind of weighted average of all the keys with the weights being the similarities between the query and each value's associated key
- Given an input sequence, queries, keys, and values are generated from the same input using 3 different linear layers
  - This is equivalent to a matrix multiplication
  - The queries and keys have the same dimension, but the values do not
  - Given an input sequence of n tokens, we will have n of each queries, keys and values
- Mathematically the attention is defined as $\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$
  - This is known as scaled dot-product attention
  - d_k is the hidden dimension of Q and K (i.e. the dimensions of the embeddings)
- Performance can be improved with *multi-head attention*: the total representation space is divided into h subspaces, parallel linear layers and attentions are run for the subspaces and the final result is the concatenation of all subspaces, after passing through another linear layer
- Each transformer encoder layer consists of a multi-head self-attention sublayer, followed by a fully connected sublayer, with residual (skip) connections around each of the sublayers followed by layer

normalization
  - Since the model has no recurrent or convolutional layers, it doesn't take into account order by itself
  - Positional encoding is used to give the model order information
    * One way to do this is alternate between sines and cosines
    * The goal is to create output in the same shape as the input word embeddings, with a unique value for each position in the sequence
    * This is then added elementwise to the input embeddings before being passed into the encoder layer
- Transformers have a number of advantages over RNNs:
  - Good with long-range dependencies (no memory loss)
  - Less likely to have vanishing or exploding gradients
  - Fewer training steps in general (due to lack of recurrent relation)
    * However for smaller datasets RNNs might be better
  - Allows parallel computation (again due to lack of recurrent relation)

Applications

- Most common application is in language processing
- Transformers can be applied to transfer learning in the same way as CNNs
- They can also be used in computer vision tasks; these are known as *vision transformers* (ViTs)
  - Compared to CNNs, ViTs achieve higher accuracies on large datasets due to their higher modelling capacity (i.e. more flexibility), lower inductive biases, and global receptive fields (i.e. they look at the entire image at once)
  - However CNNs are still on par or better in terms of model complexity or size vs. accuracy (i.e. CNNs can work better for the same number of parameters)
- ViTs split the image into smaller sections (*patches*), gives each one a positional embedding and passes it to a transformer encoder

Week 11

Deep Sets and Graphs

- Suppose we omit the positional encoding from the transformer input; then the input will be treated as a set, and representations won't change if the input tokens are shuffled
  - This is known as *order-invariance* or *permutation-invariance*
  - This is useful for certain tasks such as determining whether a molecule is toxic, where the order of input does not matter (*non-Euclidean*)
    * This is opposed to the tasks we've seen so far operating on images or text, where if the input order changes, the meaning also changes (*Euclidean*)
- In general, to achieve order invariance we first learn embeddings for each item using a shared network ψ to project into a shared latent space, then use an order-invariant aggregation function (e.g. sum, mean, max) to aggregate all input embeddings into a single embedding, and finally use another neural network ϕ to project into the output space
  - This is known as a *deep set*
- A *graph* $G = (V, E, X)$ is a data structure that encodes pairwise interactions/relations among concepts or objects as well as their features:
  - V is a set of nodes representing the concepts or objects
  - E is a set of edges connecting nodes and representing relations among them
  - X encodes the features of each node (information attached to each node)
  - The *degree* of a node is a number of edges connecting to that node
  - V and E can be represented in an *adjacency matrix* A where $a_{ij} = \begin{cases} 1 & (i,j) \in E \\ 0 & (i,j) \notin E \end{cases}$

- Graphs are order-invariant; we can arbitrarily permute the node order as long as the adjacency matrix is updated
  - Functions on graphs should also be order-invariant

Graph Neural Networks (GNNs)

- GNNs are a general type of neural network that can be modelled as a function on graphs, $f(X, A)$
  - Mostly based on *message passing*, i.e. communicating with neighbours to update embeddings
  - The inputs (X, A) are transformed to latents (H, A); the graph shape does not change but the data representation is now in latent space
  - The latents can be used for e.g. node classification (operating on each node), graph classification (operating on the entire graph), or link prediction (operating on edges)
- To perform message-passing, the embeddings of all neighbours of a node are aggregated using an order-invariant function (e.g. sum, mean, max), then combined with the node's own embedding (not necessarily order invariant), and the embedding for the node is updated
  - This is performed in parallel for every node
- To get data out of the graph, we use a *read-out* (pooling) function, which must be another order-invariant function
  - This gives us a graph embedding which we can pass to another network

Graph Convolutional Networks (GCNs)

- A GNN layer at its core is a nonlinear function over node features and the adjacency matrix, $H = f(A, X)$
- The simplest model could be $H = \sigma(AXW)$ where the activation σ provides the non-linearity and W is a weight matrix
- This has some issues we need to fix:
  - Simply multiplying with A sums up the features of neighbours but not the node itself
    - We can add self-loops, i.e. $A = A + I$
  - A is not normalized so the multiplication will completely change the scale of the feature vectors (i.e. we unintentionally make some features more important than others)
    - We symmetrically normalize A using D such that all rows sum to 1: $D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$
    - The degree matrix D is the diagonal matrix where the entries are the degrees of each node
    - We can obtain this by simply summing the rows of A and putting the result in a diagonal matrix
- The GCN layer is defined as $H = \sigma\left(D^{-\frac{1}{2}}AD^{-\frac{1}{2}}XW\right)$
- A GCN layer will only update the node embeddings based on the immediate ("1-hop") neighbours
  - To get influence from further nodes, we stack multiple GCN layers
  - This is analogous to increasing the receptive field in CNNs

Graph Attention Networks (GATs)

- Instead of using node degrees, GATs learn an attention score between two nodes
  - Similar to a transformer but without the positional encoding
- A shared neural network is used to compute the attention score between two nodes, then softmax normalized, and the node embeddings are updated based on the attention scores
  - $h_i = \sigma\left(\sum_j \alpha_{ij} W h_j\right)$ where $a_{ij} = \text{softmax}_j(e_{ij})$ where e_{ij} is the attention score between nodes i and j computed by a neural network