Lecture 1, Jan 8, 2024

Machine Learning

- *Supervised learning*: Mapping an input to an output, based on labelled data/ground truth; regression (continuous values) or classification (categorical/discrete labels)
  - Requires labelled data
  - To select the appropriate model we need to make assumptions about the problem; this is known as *inductive bias* or *learning bias*
    * "No free lunch" theorem says that we need to make assumptions to learn
    * Without proper assumptions, all models tend to perform equally if averaged over all possible tasks
  - We also need to quantify the model's performance; this is done through a *loss function*
  - More complex models have greater capacity to learn, but are more prone to overfitting – the model can learn the peculiarities about the specific training data and fail to generalize
    * To combat this, we partition the dataset into training and testing subsets; we evaluate model performance by running the model on the testing dataset
    * However with too much tuning we can also effectively overfit to the testing set
    * In practice we use 3 subsets: training (to teach the model), validation (to tune hyperparameters), and testing/holdout (to evaluate performance sparingly)
    * Ideally the final holdout set should be used only once (but this is often not possible)
- *Unsupervised learning*: learns patterns from observations without requiring ground truths
  - Includes self-supervised and semi-supervised learning
- *Reinforcement learning*: sparse rewards from the environment; actions affect the environment (e.g. training the model to play a game)

Lecture 2, Jan 15, 2024

Artificial Neural Networks

Neurons

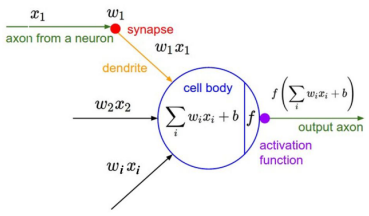
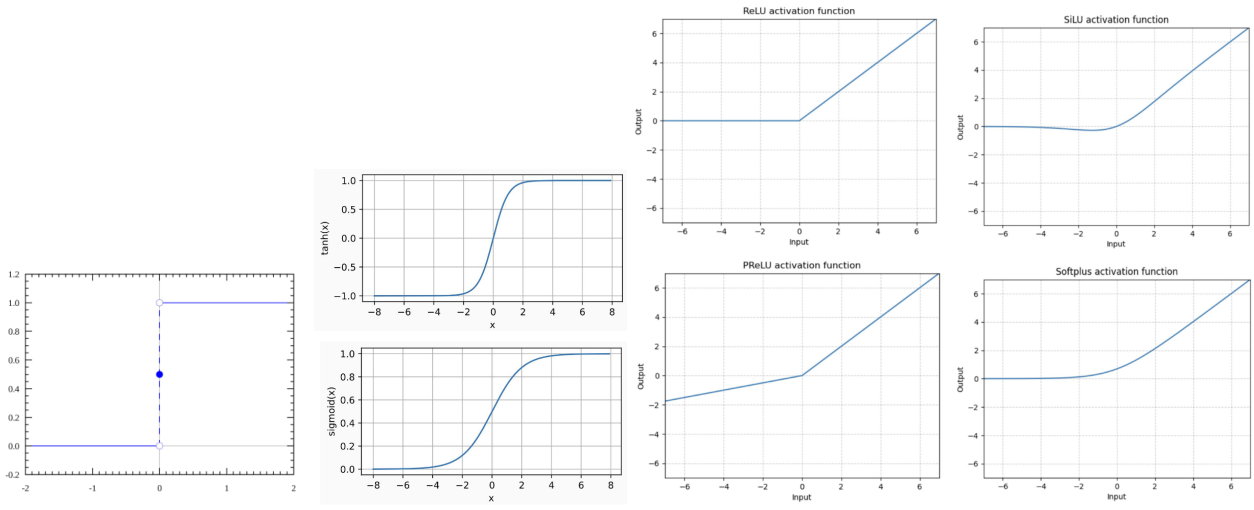


Figure 1: The artificial neuron model.

- Each neuron takes inputs $\boldsymbol{x}$, has weights $\boldsymbol{w}$ and bias b, and an activation function f which produces the output y
  - $y = f(\boldsymbol{w} \cdot \boldsymbol{x} + b)$
  - In a fully connected neural network, the inputs of the next layer are taken as all of the outputs of the previous layer
- The activation function takes the weighted sum of the input and produces an output
  - e.g. for a linear activation function $y = \boldsymbol{w} \cdot \boldsymbol{x} + b$, this draws out a hyperplane which splits the input space in 2; the $\boldsymbol{w}$ are the slopes of the plane and b controls how far it is from the origin
    * This can be used for a classification task where the data is linearly separable by drawing a line that separates the data categories
- Linear activation functions are not useful because composing any number of them will still result in another linear function, so there is no benefit to having a more complex network

  - Most data in reality is not linearly separable, so linear activation functions can never work even with many layers
- Example activation functions:
  - Perceptron: $f(x) = \text{sgn}(x) = \begin{cases} -1 & x < 0 \\ 1 & x > 0 \end{cases}$, or $f(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases}$
    * 0 is the *decision boundary*, where the output of the neuron changes
    * Used in early artificial neurons and no longer used today
    * Problem: Not differentiable, continuous or smooth
  - Sigmoid: a family including $f(x) = \tanh(x)$ (hyperbolic tangent) or $f(x) = \dfrac{1}{1+e^{-x}}$ (logistic)
    * Maps the entire range of input into an output range of $[-1, 1]$ or $[0, 1]$
    * Commonly used before 2012, still used sometimes today
    * Differentiable, smooth, and continuous
    * Problem: large inputs saturate the neuron, which kills the gradient, resulting in very slow learning; also not as performant as some other options
  - ReLU (Rectified Linear Unit): $f(x) = \max(0, x)$
    * The derivative at zero is defined as 0
    * Differentiable (and very fast to compute derivatives), continuous
    * The family also includes other functions:
      - Leaky ReLU: $x < 0$: $f(x) = \begin{cases} x & x \geq 0 \\ cx & x < 0 \end{cases}$
        - Use a small constant slope for values less than zero
      - Parametric ReLU (PReLU) makes the slope for negative values a tunable parameter for the network
      - SiLU: $f(x) = x\sigma(x) = \dfrac{x}{1+e^{-x}}$
        - Continuous approximation of ReLU
      - SoftPlus: $f(x) = \dfrac{1}{\beta}\log(1 + e^{\beta x})$
        - Another continuous approximation
        - Often gives better performance than regular ReLU but possibly slower to train



Training Neural Networks

- Training a neuron is the process of selecting the weights and bias of the neuron so the network does what we want
  - Initially the weights and biases of each neuron is randomized
  - Note we will refer to all parameters as "weights", including the bias term

- In general, training a neuron involves the following steps:
  1. Make a prediction for some input data $\boldsymbol{x}$: $y = M(\boldsymbol{w}; \boldsymbol{x})$
  2. Compare the correct output with the predicted output to compute the loss: $E = \text{loss}(y, t)$
  3. Adjust the weights to make the prediction closer to the ground truth, i.e. minimize the error
  4. Repeat until the level of error is acceptable
- Training involves a forward pass (given input, compute the output), which is used in both training and inference, and a backward pass (given the output and loss, find the effect of each weight on the loss)

Loss

- The *loss function* is a measure of performance of the network; it computes how bad predictions are compared to ground truth labels
  - The larger the loss, the worse the network's performance is
  - We want to compute the loss over all the input data and take the average
- To compare against the ground truth label, we first have to convert the label and the output of the network into matching forms
  - A *softmax* function normalizes the network output into a categorical probability distribution; this is used for single-label classification tasks
    * $\text{softmax}(x_i) = \dfrac{e^{x_i}}{\sum_{k=1}^{K} e^{x_k}}$
    * This converts the non-normalized output from the network into a probability distribution that sums to 1
    * The network's output before passing through any activation is called *logits*
  - Then use a *one-hot encoding* to map category labels to a vector representation; the element representing the category of a label is 1, while all other labels are 0
    * This can also be interpreted as a probability distribution
- Example loss functions:
  - Mean squared error (MSE): $\dfrac{1}{N}\sum_{n=1}^{N}(y_n - t_n)^2$
    * N training samples, with network predictions y_n and true labels t_n
    * Used mainly for regression tasks because it doesn't work well with probabilities
  - Cross entropy (CE): $-\dfrac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} t_{n,k}\log(y_{n,k})$
    * N training samples, K classes ($t_{n,k}$ is the probability of training sample n being in category k)
    * Used for classification tasks since it works on probabilities
  - Binary cross entropy (BCE): $\dfrac{1}{N}\sum_{n=1}^{N}\left(t_n\log(y_n) + (1-t_n)\log(1-y_n)\right)$
    * Used for binary classification tasks, where the output can either be 0 or 1
    * A special case of the cross entropy loss function

Gradient Descent

- Ultimately training a neural network is an optimization problem; we want to find the minimum of the loss function by adjusting the weights of the network
  - This can be accomplished using gradient descent
- When training we want to find how changing each weight of the neuron affects the final output, i.e. finding $\dfrac{\partial E}{\partial w_{ji}}$
- Once we find the gradient, the weights are updated as $w_{ji}^{t+1} = w_{ji}^{t} - \Delta w_{ji} = w_{ji}^{t} - \gamma\dfrac{\partial E}{\partial w_{ji}}$
  - γ is the *learning rate*, or step size of the gradient descent
  - In the most simple case, γ is set to a constant (adaptive size methods also exist)

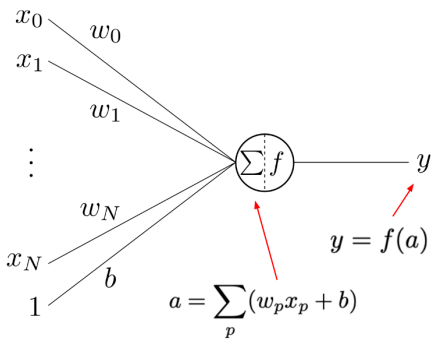


Figure 2: Setup for example problem.

- Example: consider MSE loss $E = (y - t)^2$ with sigmoid activation $f(x) = \dfrac{1}{1 + e^{-x}}$; how do we compute $\dfrac{\partial E}{\partial w_p}$?
  - Using the chain rule: $\dfrac{\partial E}{\partial w_p} = \dfrac{\partial E}{\partial y}\dfrac{\partial y}{\partial a}\dfrac{\partial a}{\partial w_p}$
  - $\dfrac{\partial E}{\partial y} = \dfrac{\partial}{\partial y}(y - t)^2 = 2(y - t)$
  - $\dfrac{\partial y}{\partial a} = \dfrac{\partial}{\partial a}\left[\dfrac{1}{1 + e^{-a}}\right] = y(1 - y)$
  - $\dfrac{\partial a}{\partial w_p} = \dfrac{\partial}{\partial w_p}\left[\sum_p w_p x_p + b\right] = x_p$
  - Multiplying this together: $\dfrac{\partial E}{\partial w_p} = 2x_p(y - t)(1 - y)y$
- The gradients are easy to find for layers that are next to the output, but for intermediate layers this is requires *backpropagation*

Network Architecture

- Having a single decision boundary is insufficient for most problems, so having multiple layers is necessary
- As the number of layers approaches infinity, a neural network approaches a universal function approximator
- However for deeper networks, computing the gradient is harder
  - The problem of finding these gradients is the *credit assignment problem* – how much influence does each weight have on the error?
  - This is solved by backpropagation
- With multiple layers, we can think of each layer picking out features of the data that get higher and higher in level with deeper layers
  - The complex, non-linearly separable data is processed by earlier layers into a form that is linearly separable at the final output layer
- *Feed-forward network*: information only flows forward from one layer to a later layer, from input to output
- *Fully-connected network*: each neuron takes its input from all neurons in the previous layer; i.e. Neurons between adjacent layers are fully connected
- The total number of layers is the number of hidden layers plus the output layer
  - We don't count the input layer (because it's decided by the input data format, so we don't have control), but we do count the output layer

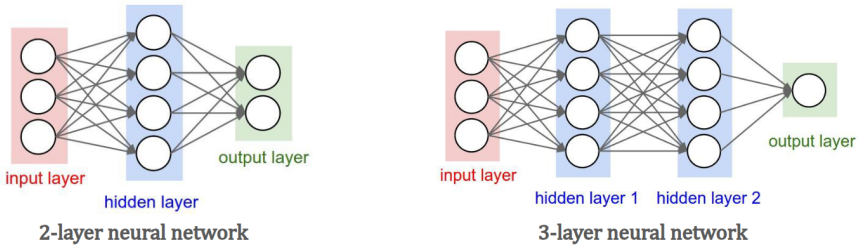


Figure 3: Example 2- and 3-layer neural networks.

Lecture 3, Jan 22, 2024

Hyperparameters

- Hyperparameters are parameters that are not optimized during learning
- This can include:
  - Batch size
    - Rule of thumb: start out with a batch size that will max out the GPU
  - Number of layers
  - Layer size
  - Type of activation function
  - Learning rate
  - etc.
- Weights are updated through gradient descent as the inner loop of optimization; hyperparameters are tuned in the outer loop of optimization
- The full dataset (historic data) is split into train, validation, and test dataset (stratified sampling without replacement)
  - The training dataset is used to train the model itself
  - The validation dataset is used to tune the hyperparameters
  - The testing dataset is only used to evaluate the final model

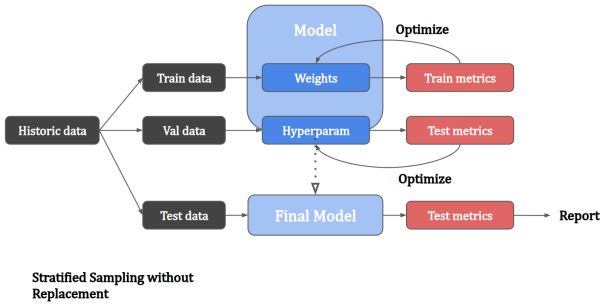


Figure 4: Stratified sampling without replacement.

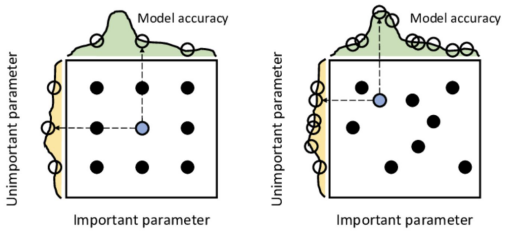


Figure 5: Grid vs. random search for hyperparmeter tuning.

- To tune hyperparameters, we can use either a grid search or a random search
  - In a grid search, we search over a grid of evenly/regularly spaced values for each hyperparameter
  - In a random search, we pick the combinations of hyperparameters randomly
  - Both have the problem that the number of points we need to fill the space increases exponentially with more hyperparameters (curse of dimensionality), so for larger problems it may become infeasible to search the entire space
  - Other techniques such as Bayesian optimization also exist

Optimizers

- Defining a loss function turns our learning problem into a mathematically defined optimization problem
- An optimizer determines how each parameter should change to minimize the loss function
- All the optimizers that we use in this course are based on *gradient descent*
- Pytorch automates the gradient computation

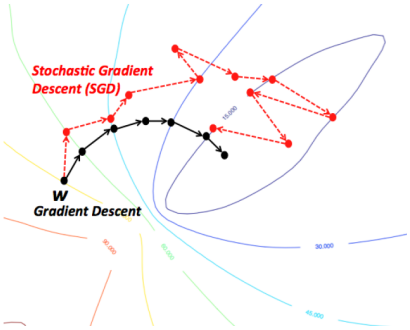


Figure 6: Comparison of path taken by regular vs stochastic gradient descent.

- Stochastic gradient descent (SGD)
  - In SGD, in each iteration we evaluate a training sample taken from the dataset at random
  - Computing the gradient takes less time since the sample is smaller, but overall this may not be faster
  - Gradient descent takes into account the entire dataset at a time while SGD only look at a subset
    * This means SGD is only approximating what the overall gradient looks like
    * As a result, instead of a smooth path to the minimum, the optimization looks more erratic and jumps around a lot more
  - However, SGD allows you to do a more global search, and often results in a better set of weights
- Mini-match gradient descent
  - Instead of working with a single sample at a time, we batch samples together
  - Use the network to make predictions for n samples, average the loss for these samples, and take a step to optimize the average loss for these samples
  - This procedure is very widely used
  - Definitions:
    * Batch size: number of training samples used per optimization step
    * Iteration/step: each time we change the weights of the model
    * Epoch: number of times that we go though all the training data
  - e.g. if there are 1000 samples in the training data, with a batch size of 10, each epoch will have 100 iterations
  - When plotting loss, we usually plot against the epochs instead of iterations
  - If the batch size is too small, we are optimizing a potentially very loss function at each iteration, so the result is noisy
  - If the batch sizes are too large, each iteration is very expensive and the average loss might not change a lot as the batch size grows
    * Larger batches are not always better (overfitting)

- Since we're optimizing in a large number of dimensions, many weights end up at saddle points which have zero gradient
  - Plateaus are are problem but can be addressed using specialized variants of gradient descent, such as momentum

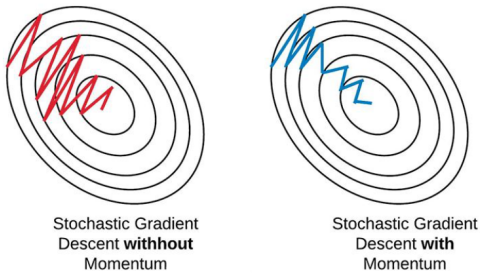


Figure 7: SGD with vs. without momentum in a ravine.

- *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
- SGD with *momentum* remedies this by adding a momentum term
  - $v_{ji}^t = \lambda v_{ji}^{t-1} - \gamma \frac{\partial E}{\partial w_{ji}}$
  - $w_{ji}^{t+1} = w_{ji}^t + v_{ji}^t$
  - $\lambda = 1 - \gamma$
  - Note that we have no way of changing the momentum; if we make a bad choice at the beginning, we're stuck with it
- In *adaptive moment estimation* (Adam) each weight has its own learning rate; this incorporates both momentum and adaptive learning rate
  - $m_t = \beta_1 m_{t-1} + (1 - \beta_1) \frac{\partial E}{\partial w_{ji}}$
  - $v_t = \beta_2 v_{t-1} + (1 - \beta_2) \left(\frac{\partial E}{\partial w_{ji}} \right)^2$
    - Squaring the gradient emphasizes it
  - $w_{ji}^{t+1} = w_{ji}^t - \frac{\gamma}{\sqrt{v_t} + \epsilon} m_t$
    - The ϵ is a noise term so that we won't divide by zero
  - Advantages include rapid convergence, minimal tuning
  - Note in Pytorch the β_1 and β_2 parameters are set to the defaults found in the paper
  - We can use this in pretty much all situations

Learning Rate

- The learning rate γ is the size of the step that an optimizer takes during each iteration
- A larger step size changes the parameters more in each iteration
- A good choice would be 0.01 to 0.001
  - If the learning rate is too large, it can overshoot or bounce around the minimum
  - If it is too small, it will take a very long time
- The appropriate learning rate depends on the learning problem, type of optimizer, the match size and the stage of training
  - Larger batches require larger learning rates (since there are fewer iterations)
  - As the stage of training progresses the learning rate should be reduced
  - Methods of adjusting the learning rate can include step decay, exponential decay and two-stage exponential decay

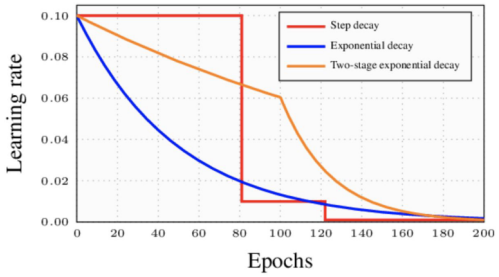
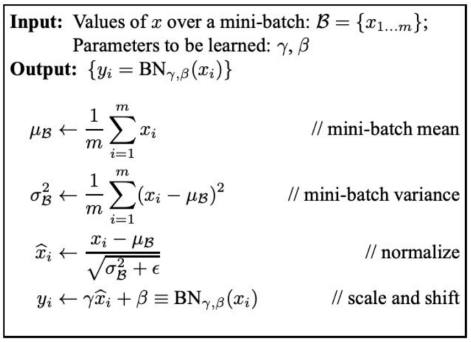


Figure 8: Methods of adjusting the learning rate.

Normalization

- We normalize the inputs to prevent the model from paying attention to the scale of features
  - Without normalization, the model would pay more attention to features with larger range
  - Always normalize the inputs!
- The inputs are normalized as $X_i = \dfrac{X_i - \mu_i}{\sigma_i}$ across all inputs
  - This only normalizes the data for the first layer; we still need to normalize subsequent layers



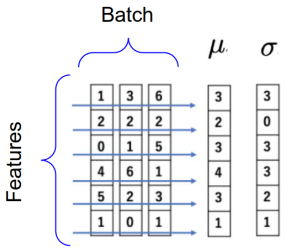


Figure 9: Batch normalization.

- Batch normalization: normalize activations batch-wise for each layer, done right before activation, but after multiplying by weights and adding bias
  - Again we add some amount of noise ϵ to prevent dividing by zero
  - This "centers" the data around the active region of the activation function (around zero), which speeds up training
  - After normalization, we can scale and shift the data as well (not used too often)
    - This adds more parameters to be learned
  - We still need to normalize during inference time since the network is used to seeing normalized data
    - Use a moving average across the entire training dataset
      - $\mu_{mov_i} = \alpha\mu_{mov_i} + (1-\alpha)\mu_i$
      - $\sigma_{mov_i} = \alpha\sigma_{mov_i} + (1-\alpha)\sigma_i$
    - At inference time, use this moving average from the training data to normalize
  - Advantages:
    - Allows higher learning rates, which speeds up the training
    - Regularizes the model
    - Less sensitivity to the initialization of the model (initially weights are randomly chosen)
  - Disadvantages:
    - Depends on the batch size; has no effect with small batches
    - Can't work with SGD (since summary statistics are computed over the batch, so SGD with

batch size of 1 is useless)

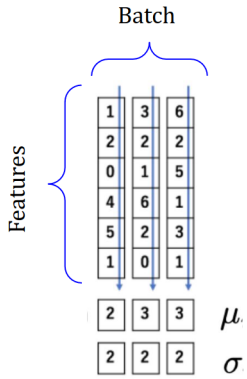


Figure 10: Layer normalization.

- Layer normalization: normalize across all the neurons for an entire layer
  - Much simpler to implement and we don't need moving averages or parameters
  - Not as effective as batch normalization however

Regularization

- Regularization are techniques that reduce overfitting or underfitting
- Dropout: randomly set weights to 0 with probability p (i.e. drop out connections)
  - Every time we train, the weights that are dropped out are shuffled
  - This forces a neural network to learn more robust features since the model is forced to work with less data
  - During inference, multiply all weights by $(1-p)$ to keep the same distribution
  - Dropouts are performed after activation
- *L2 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 $W_{t+1} = W_t - \gamma \left(\alpha W_t + \frac{\partial E}{\partial W} \right)$
  - Prevents the weights from exploding

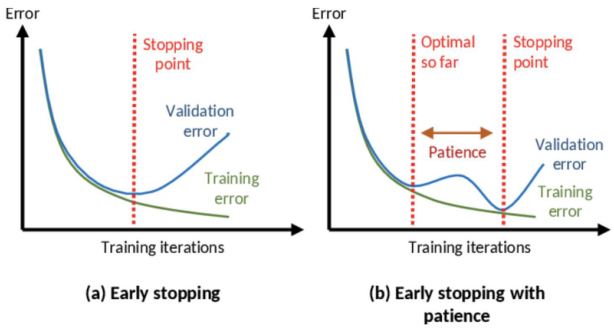


Figure 11: Early stopping with patience.

- *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
  - This a common technique; we often have to wait multiple times

PyTorch Implementations

- Use `torch.manual_seed()` to ensure reproducibility, since model weights are initialized randomly
- Class `nn.Linear(in, out)` defines a fully connected linear layer
- To define a network, we inherit from class `nn.Module`
  - In the constructor we set up the network
  - In the `forward()` method we do a forward pass to make a prediction
    * We take the input and pass through the layers by calling them as functions
    * To apply activation functions we call methods from `nn.functional`, e.g. `relu()`
    * Note we don't apply an activation function for the final layer
      - PyTorch has implementations of loss functions that include the final activation
        - `nn.BCEWithLogitsLoss()` applies sigmoid internally, whereas `nn.BCELoss()` does not
        - `nn.CrossEntropyLoss()` applies softmax internally, whereas `nn.NLLLoss()` expects a proper distribution
      - For numerical stability these are better
- Calling the model like a function allows us to do the forward pass
- To compute gradients, we call `backward()` on the loss function (e.g. `nn.BCEWithLogitsLoss`, `nn.CrossEntropyLoss`)
- Use an optimizer in `optim` and specify parameters; we also need to pass in the tunable parameters via `model.parameters()`
  - For each sample in the training dataset:
    1. Make a prediction by calling `model(input)`
    2. Compute the loss by calling `criterion(out, labels)`
    3. Compute gradients using `loss.backward()`
    4. Update parameters by calling `optimizer.step()`
    5. Clean up optimizer by calling `optimizer.zero_grad()` (otherwise the optimizer saves previous gradients)
- For multi-class classification, the output layer needs to have as many neurons as classes, we need to softmax the final layer and use multiclass cross-entropy loss

Debugging Neural Networks

- Make sure the model can overfit on a small amount of data
  - Do this with a small amount of data to do it quickly
- Ensure that the model is training at all – check that loss is actually going down

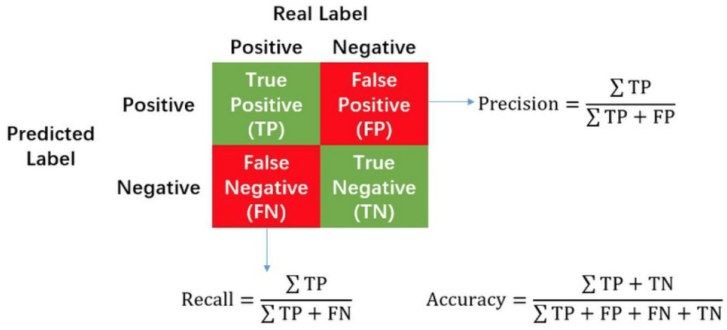


Figure 12: Confusion matrix.

- The confusion matrix can be used to check the model performance and the data balance
  - 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*: $2 \times \frac{\text{recall} \times \text{precision}}{\text{recall} + \text{precision}}$
- We can also take data from later layers and plot the data to see if there are patterns
  - Using t-SNE for a 2D projection and visualization of the data structure
  - We should expect to see that different classes are clumped together

Lecture 4, Jan 29, 2024

Convolutional Neural Networks (CNNs)

- Using a regular ANN has disadvantages:
  - By flattening the image we lose geometric information about what pixels are next to each other
  - We are restricted to a specific image size (need to retrain the entire model if we change it)
  - The data needs to be preprocessed in a specific way (e.g. centered)
  - Computational complexity grows very quickly as layers get bigger

The Convolution Operator

- A *convolution* is an operation that slides a *kernel* across an image, taking a weighted sum of the part of the image overlapping with the kernel for every kernel position
- $y[m,n] = I[m,n] * K[m,n] = \sum_{j=-\infty}^{\infty} \sum_{i=-\infty}^{\infty} I[i,j]K[m-i, n-j]$
- Convolutional filters can achieve various effects on an image, including blurring, edge detection, etc
- Kernels used to be hand-crafted, but in a CNN we make the network learn the kernel
- Applying a convolution to an image reduces the size of the image, unless we apply *padding* to the edges
  - We can add zeroes around the border to make the output the same size, or even bigger if we desire
  - Our feature space retains the same dimensionality and we don't lose any information around the edges

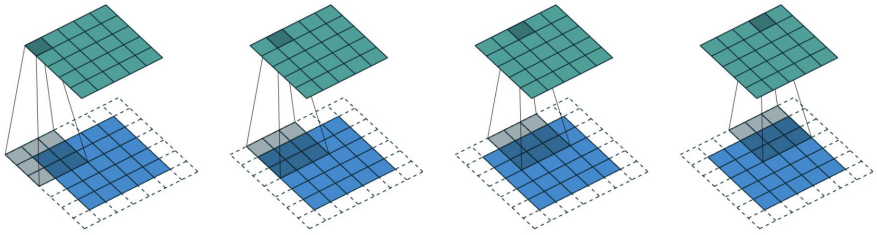


Figure 13: Illustration of padding.

- We can also change the *stride*, or how much the kernel moves each time
  - Increasing the stride reduces the output resolution and can act as a form of pooling
  - Lowering the output dimension can lower the number of parameters we need to learn
- Each output 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
  - Note different dimensions might have different amounts of padding, stride, etc

Convolutional Neural Networks

- Use convolutional filters in the networks, where the kernels are learned by the network
- CNNs use locally connected layers (kernels act on a small, local region of the image) and use weight sharing (the same local features are detected across the entire image)
  - This retains the geometric information in the image that would otherwise be lost by flattening
  - Weight sharing significantly reduces the number of parameters that need to be learned

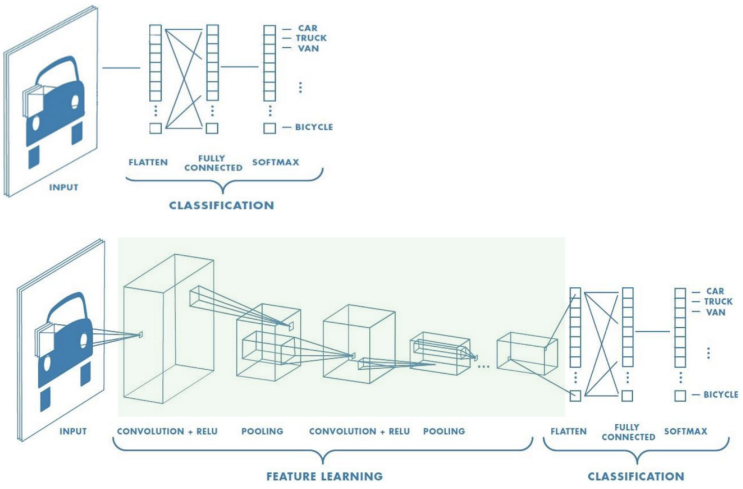


Figure 14: Illustration of CNNs vs ANNs for classification.

- The later layers will learn more abstract/higher level features and there will be fewer neurons
  - At the end we flatten the features and pass to an ANN for classification
  - At this point the features are very abstract and no longer geometric, so we don't lose information
  - The CNN layers are the *encoder*, which extracts features from the image, while the ANN layers are the *classifier* or *head*, which classifies the image based on features
- The network learns all the weights in the kernel, as well as a bias for each kernel
  - The weights are randomly initialized
- Images and convolutional layers can have multiple channels
  - 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 (increasing the *filter depth*)
    - The number of kernels is the number of output channels – each kernel produces its own output channel
    - Each kernel will learn a different set of features because they are randomly initialized, so upon gradient descent they will move towards detecting different features
  - 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 layers, the filter depth increases, and the feature map size decreases; i.e. we have more sets of features that are each individually lower in resolution

Pooling

- *Pooling* is a way to consolidate information, i.e. removing information not useful for the task
  - This is like reducing the layer size before the final output layer in an ANN
- Pooling is essentially another convolution over the output, but the kernel is not learnable:
  - *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

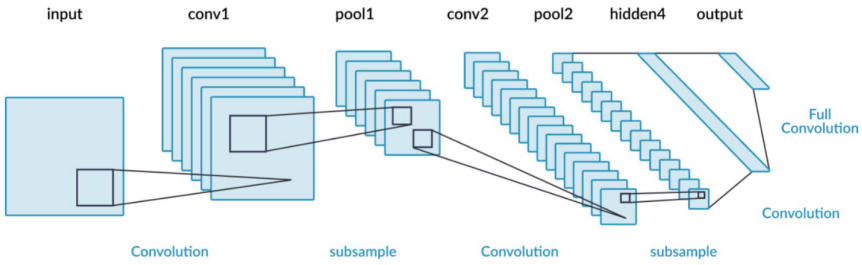


Figure 15: Progression of convolutional layers in a network.

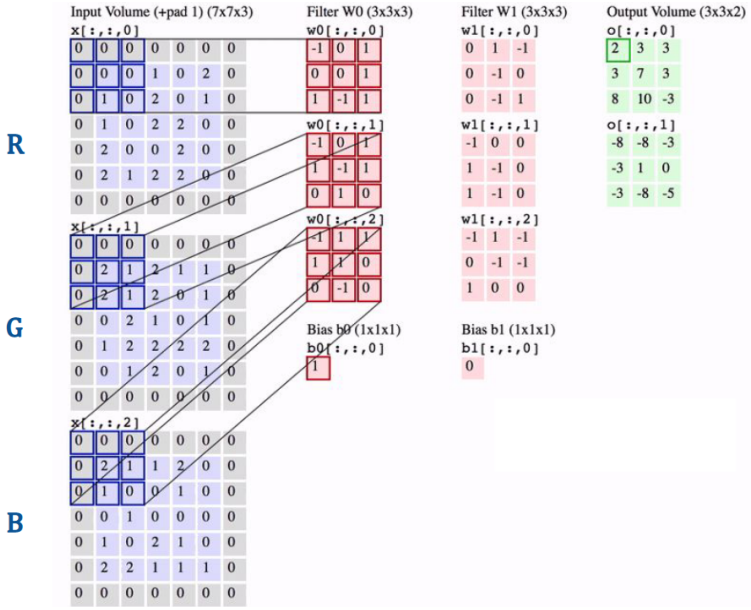


Figure 16: Convolution for $3 \times 5 \times 5$ input (1 padding), with $2 \times 3 \times 3$ convolutional kernels.

  - Output dimension is given by $o = \left\lfloor \frac{i-k}{s} \right\rfloor + 1$
- An alternative to pooling is to just use another convolution layer with a larger stride
  - Since this kernel can be learned, it introduces more parameters
  - This makes the model more powerful but increases computational cost

PyTorch Implementation

- Use `nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding)` to implement a convolutional layer
  - Default for stride is 1, padding is 0
  - Specify integers to use the same across 2 dimensions, or make it a tuple for different parameters in each dimension
- Use `nn.MaxPool2d(kernel_size, stride)` etc for pooling
- Once the convolutional layers are done we go back to using `nn.Linear()` to implement the ANN layers
- First apply the convolutional layer, then the activation function, then the pooling
- The training code stays the same whether it's a CNN or ANN because PyTorch handles all the gradient calculations

Lecture 5, Feb 5, 2024

Convolutional Neural Networks (Continued)

Visualization of Convolutional Filters

- We can display the convolutional filters as images to get an idea of what the network is learning
  - The earlier layers have filters that capture low-level features such as edges and blobs, similar to hand-crafted features
  - Later layers pick up more abstract features, and at this point we can't tell what the filters do anymore
- One way to visualize which parts of the image the network picks out is to use a *saliency map*
  - Pass the input image through the network and compute the loss
  - Take the gradients with respect to the inputs (as opposed to the weights as is during training)
  - For multiple channels, take the max absolute value across all channels for each pixel
  - Plotting this will give us a heatmap of which regions the network considers relevant to the problem
- However saliency maps are not very practical because they don't give much detailed information
  - This can also be misleading

Evolution of CNNs

- LeNet was the original CNN introduced in 1989; we mostly refer to LeNet-5
  - LeNet demonstrated invariance to translation, scaling, rotation, squeezing/stretching, stroke width, and noise
- For the next 20 years CNNs were outperformed by other techniques such as random forests/decision trees or support vector machines
- *Deformable parts models* were one popular technique used at that point
  - Detect features that make up the object and their locations
  - Introduce constraints on where the features can be in relation to each other
  - Allow some amount of deformation before the object is considered invalid
  - However this requires handcrafting the models for each type of object; models are not transferable or generalizable
  - Also only works well with one specific perspective/view angle
- ILSVRC is a subset of the ImageNet dataset and is a competition for image recognition models
- AlexNet was an entry of ILSVRC that significantly improved over past results

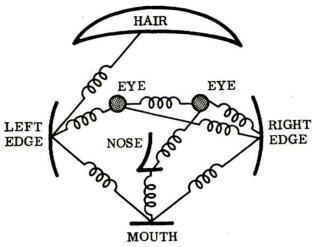


Figure 17: Illustration of a deformable parts model for a face.

  - Significantly larger than LeNet – more parameters and much deeper
  - Used ReLU instead of sigmoid
  - Used dropout, weight decay, scheduled learning rates and data augmentation
- GoogLeNet achieved near-human accuracy in 2014
  - Much deeper than other networks with 22 convolution layers
  - Only 4 million parameters vs. 60 million for AlexNet
  - Increased parameter efficiency was achieved using *inception blocks*
- ResNet achieved better-than-human accuracy in 2015 using skip connections, batch normalization, and ReLU
  - The use of skip/residual connections allowed the network to be very deep (152 layers)
  - Used strided convolutions instead of pooling layers
  - Only a single fully-connected classification layer since the convolution layers were very good
  - Uses global average pooling
- ILSVRC is now a solved problem
- Object recognition in the wild is still an open problem

Modern CNN Techniques

- *Data augmentation*: applying class-preserving transformations to the input to generate more data, e.g. cropping, resizing, translation, color filtering, noise addition, blur
  - PyTorch has tools to do this
  - Helps generate more data and increase robustness
- Increasing the depth of models improves generalization performance, but with very deep models, we run into issues of vanishing or exploding gradients due to the long path the input takes to the loss function
  - To address this we can use better initialization for ReLU, skip connections, and normalization
- *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; this significantly improves parameter efficiency

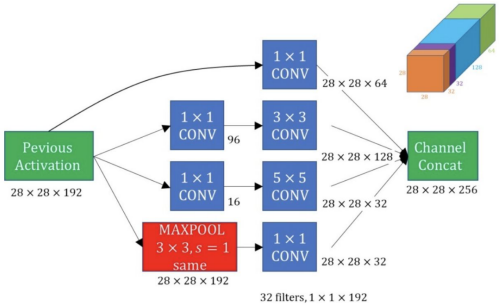


Figure 18: Inception block concatenating convolutional filters of multiple different sizes.

- *Pointwise convolutions*: Applying pixel-wise transformations that map each pixel into a higher or lower dimensional space, e.g. turning RGB pixels into a single value
  - These transformations are nonlinear due to the activation function
  - Used in most modern CNN architectures except VGG
  - This is useful for consolidating information between channels

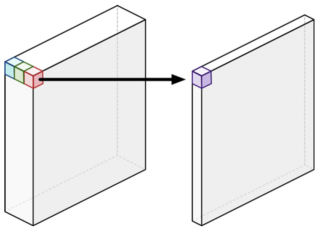


Figure 19: Illustration of pointwise convolutions.

- *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
  - During inference the intermediate classifiers are discarded
  - This helps with the vanishing and exploding gradient problem and can also help with overfitting
  - At the time, skip connections didn't exist

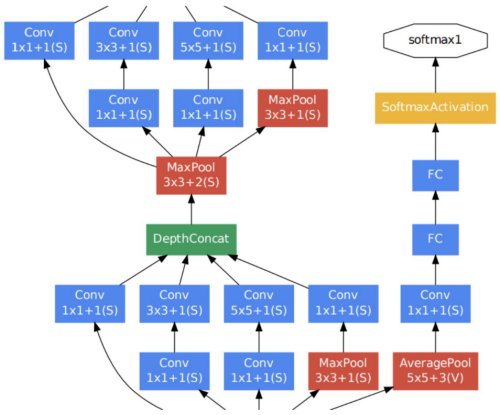


Figure 20: Intermediate classifier with auxiliary loss.

- *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 a smaller sized filter multiple times to approximate a larger sized filter with only 3×3 filters
  - This makes hyperparameter tuning easier; we no longer have to consider different kernel sizes, just the number of filter layers
- VGG also introduced a data augmentation scheme commonly used today
- *Residual networks* (ResNets): using skip connections to provide deeper layers more direct access to outputs of earlier layers

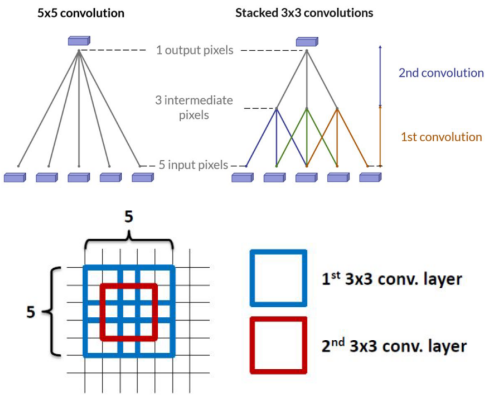


Figure 21: Stacked convolutions architecture from VGG.

  - 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
  - Effectively introduces a shortcut for the data/gradient
  - This helps with vanishing gradients and allows training very deep networks

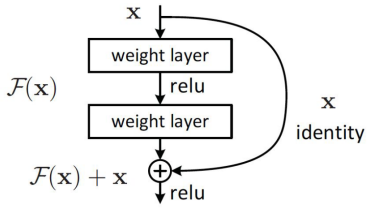


Figure 22: Skip connections in a residual network.

- *Global average pooling*: instead of flattening the final feature maps, take an average cross the entire feature map for each channel
  - The result is fed to ANN layers or directly to softmax
  - This effectively reduces each feature map's dimensions to just 1×1
  - To implement in PyTorch we can use an `nn.AdaptiveAvgPool2d(1)` layer
    * Argument is the desired output size, which is 1 for GAP
    * Kernel size is computed automatically, which makes it average across the entire input

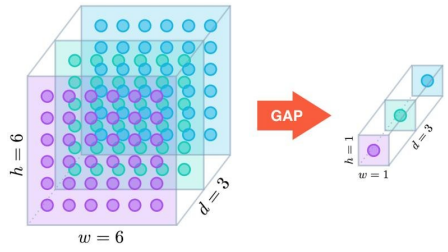


Figure 23: Illustration of global average pooling.

Transfer Learning

- 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
  - The encoder essentially becomes a universal feature extractor
- 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 such as AlexNet 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*
    * Start by tuning the later layers first
    * If the dataset is big enough we can try to tune the earlier layers
  - This helps the CNN adapt to the new task
- Models are available pre-trained from `torchvision`
  - Use `torchvision.models` to create the models, and pass `pretrained=True` to get the pre-trained weights

Lecture 6, Feb 12, 2024

Unsupervised Learning

- Supervised learning requires large amounts of labelled data, which is expensive to obtain
- In *unsupervised learning*, we look for patterns in the data without being explicitly provided labels
  - e.g. clustering, probability density estimation, dimensionality reduction
- With *self-supervised learning*, the labels are generated automatically form the data
  - e.g. masking out a part of an image and getting the model to fill it in
- With *semi-supervised learning* the data mostly consists of unlabelled samples, but a small subset is labelled

Autoencoders

- Autoencoders aim to find efficient representations of the input that contains enough information to reconstruct it
- Consists of two components:
  - *Encoder*: converts the input to an internal *embedding*, i.e. a lower dimension representation
    * Performs dimensionality reduction
  - *Decoder*: converts the embedding back to the same dimensionality as the input
    * This is a generative task
- The network has a sideways hourglass shape, with layers getting progressively smaller until we reach the *bottleneck layer*, and then getting bigger until we match the input dimension
  - All the information from the input is squeezed through the low-dimensional bottleneck layer
  - By introducing this low-dimensional layer, the model is forced to learn only the most important parts of the input and drop unnecessary features
  - The choice of the number of neurons in this layer is an important parameter
  - If the bottleneck layer is too small, not enough information will be retained to reconstruct the input
  - Autoencoders are often symmetric, but this is not a requirement
- To train these models, we use an MSE loss (`nn.MSELoss`) and compare the output against the input
- Common applications:
  - Feature extraction
  - Unsupervised pre-training
    * The encoder brings the data into a (more) separable form

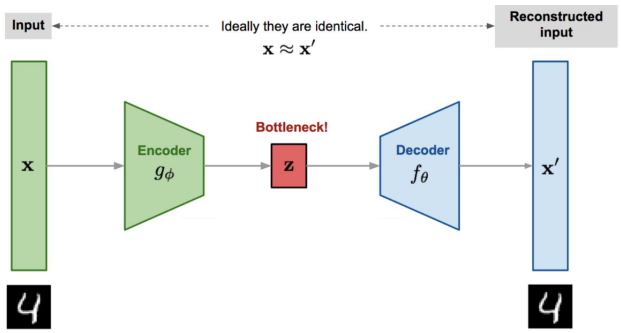


Figure 24: Illustration of an autoencoder.

    * Using the encoder and attaching a classifier to it for classification tasks
  - Dimensionality reduction
  - Generating new data
    * Sampling in the latent space and using the decoder to generate data
  - Anomaly detection
    * Autoencoders are bad at reconstructing outliers
    * If the autoencoder generates nonsensical output, there's a high chance the input is an outlier
- Compare the input and its reconstruction generated by the model to assess the model performance
  - Perfect reconstruction can be a sign of overfitting
- We can add noise to the input image and make the model reconstruct the image without noise
  - This forces the model to only learn useful features
  - This prevents the autoencoder to simply copy its inputs, so it helps with overfitting
- We can explicit the structure in the embedding space and sample from it in order to generate new data
  - This relies on the network mapping similar inputs to similar embeddings
  - The simplest way to do this is to interpolate between the embeddings of two known inputs
    * e.g. passing two numbers through the encoders, interpolating between the embeddings and passing this through the decoder to obtain an image between the two numbers
- However, if we just sample a random point in the embedding space, we will likely get a nonsensical result
  - The embedding space can become disjoint and non-continuous

Variational Autoencoders (VAEs)

- Addresses the issues with generating nonsensical results by imposing a constraint on the latent space so that it becomes smooth
  - Can be thought of as an autoencoder that is trained so that the latent space is regular enough for data generation
- 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
  - The encoder will give a mean vector and covariance matrix as its output, which encodes the distribution
  - Practically to obtain the input to the decoder, we sample a deviation $\boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{1})$, scale this up by the variance, and add it to the mean to produce a sample from the latent space
  - This allows us to compute the gradient by regarding $\boldsymbol{\epsilon}$ as a constant
- 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

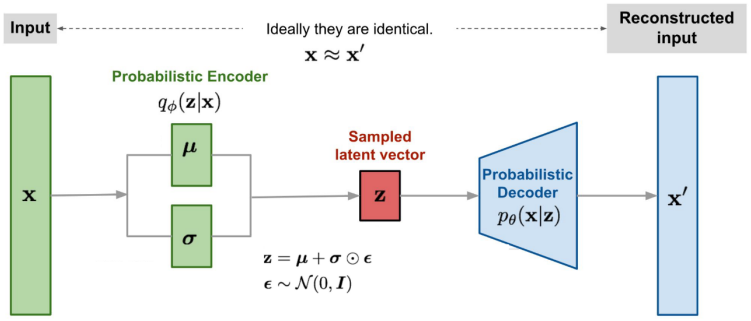


Figure 25: Illustration of a variational autoencoder.

variances (so the output is essentially a fixed point), or having very different means (so regions corresponding to different inputs are very far apart); 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 DL divergence (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
- In PyTorch this is performed using `nn.ConvTranspose2d(in_channels, out_channels, kernel_size)`
  - For the same parameters, passing the result of `nn.Conv2d()` to `nn.ConvTranspose2d()` or vice versa will give back the same shape

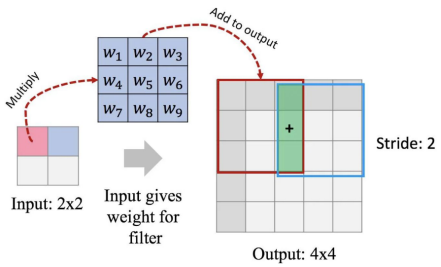


Figure 26: Transposed convolutions.

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
- This allows for semi-supervised learning; use the unlabelled data to train the autoencoder, and then use the labelled data to train the classifier
  - Since the classifier is smaller and will have access to the pre-trained encoder, it will require far less data to train

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
    - This forces the model to learn patterns of the input that stayed the same despite the augmentations
  - The other samples are generated through data augmentation methods from the original samples, e.g. inverting the image, rotating it, etc
  - Unlike autoencoders, the embeddings generated are not used to reconstruct the input, but instead used for discriminating between samples
- Encoder layers trained through self-supervised learning can be used in transfer learning

Lecture 7, Mar 4, 2024

Processing Words

- We want to convert words/symbols into structured embedding space, such that similar words are close together
- 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:

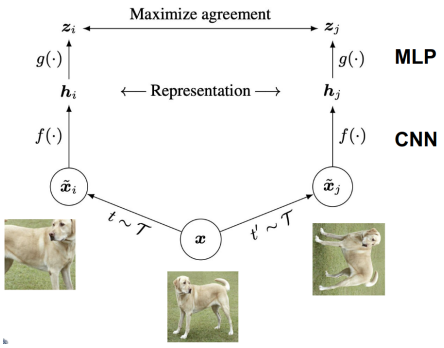


Figure 27: SimCLR architecture for contrastive learning.

    * Number assignments are arbitrary and introduces bias
    * Network is bottlenecked to a 1 dimensional representation, limiting its capacity to learn
  - The embedding space will have a much smaller dimensionality so we can actually capture meaning
- Given a word we want to find its embedding in latent space, such that similar words are close together, and math operations can be performed over embedding vectors such that the result is meaningful
  - We can't simply use an autoencoder to reconstruct the input because we don't accomplish anything
  - Important to note that the meaning of a word comes from its context
  - word2vec and GloVe are two commonly used models
- word2vec is a family of two architectures: skipgram and CBOW (continuous bag of words)
  - Skipgram takes a word (or multiple words) 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
    * Note the input and output are both a single word (not multiple at a time)
    * Advantages:
      - Works well with smaller datasets
      - Better semantic relationships (e.g. relating "cat" and "dog")
        - This is because surrounding context is based on the meaning of words
      - Better representation of words that appear less frequently
  - CBOW does the opposite and predicts the centre word(s) from context
    * In this case the input is multiple words
    * Advantages:
      - Trains faster since the task is simpler
      - Better syntactic relationships (e.g. relating "cat" and "cats")
        - This is because possible centre words are based on syntactic information from its context
      - Better representation of words that appear more frequently
  - In both architectures we use a sliding window over the text; this doesn't have to move continuously (we can put the window in random places)
  - 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}
  - When training this basically adds a term to the loss function for the correlations
- To compare similarity of word embeddings, we have 2 common choices of distance measure:
  - Euclidean distance: L2 norm of difference of embeddings

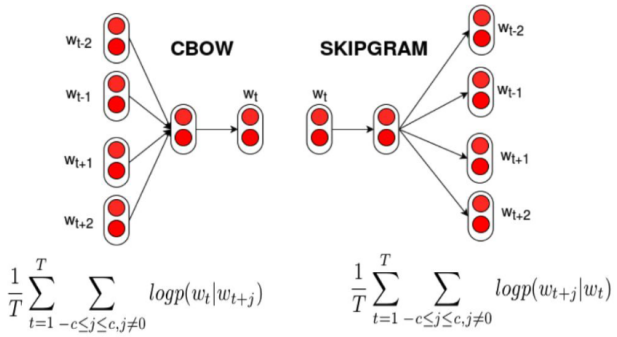
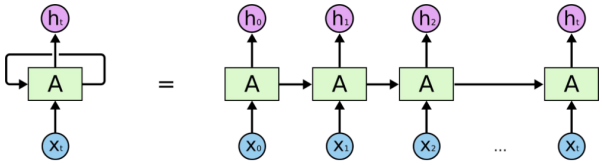


Figure 28: Illustration of skipgram and CBOW architectures.

    * $\sqrt{\sum_{i=1}^{d}(x_i - y_i)^2}$
    * Use `torch.norm(a - b)`
  - Cosine similarity: cosine of the angle between embeddings (equivalent to dot product of the embeddings divided by product of their magnitudes)
    * $\frac{\sum_{i=1}^{d} x_i y_i}{\sqrt{\sum_{i=1}^{d} x_i^2}\sqrt{\sum_{i=1}^{d} y_i^2}}$
    * Use `torch.cosine_similarity(a.unsqueeze(0), b.unsqueeze(0))` (since cosine similarity expects matrices, we need to add an extra dimension)
- As with transfer learning, we often use pre-trained models to generate embeddings
  - `torchtext` contains pre-trained GloVe embedding
  - The 6B model was trained on the 2014 Wikipedia corpus
  - Pretrained models with different embedding sizes are available

Recurrent Neural Networks (RNNs)

- Processing text is difficult because the input size is variable
  - Once we have the embeddings for each word, we still need a way to pass entire sentences to the network at a time
  - Simply concatenating all the embeddings leads to very large inputs, wasted space and bias
  - Using a convolutional filter won't be able to capture long-term dependence or global information
  - We need an architecture that allows us to remember previous information through memory
- 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
  - The size of the hidden state are possibly different, but after transformation by the weight matrices, they should be the same size
- 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
  - Hidden states are initialized to zero before the first word input
- In PyTorch, use `nn.RNN(input_size, hidden_size, batch_first=True)`
  - Calling the RNN layer requires us to pass an initial hidden state, which we can just pass as all zeros
    * Generating can be done with `torch.zeroes(1, x.size(0), hidden_size)`
    * The first dimension is the number of layers, the second is the batch size, the third is the hidden size
  - Look up embeddings using `nn.Embedding.from_pretrained(glove.vectors)` (call the resulting



$$h_t = \sigma_h(W_h x_t + U_h h_{t-1} + b_h)$$
$$y_t = \sigma_y(W_y h_t + b_y)$$

Figure 29: Illustration of the structure of an RNN.

object as a function)
- The output returns a tuple, where the first element is all the outputs and the second is the final hidden state
- The output's middle dimension are all the outputs, including intermediate ones, so we only take the last one, with `output[:, -1, :]`
  * The first dimension is the batch size; the last dimension is the hidden size
- Note that this will give us the raw hidden state outputs, so we need to pass it through additional fully connected layers to extract the output

Lecture 8, Mar 11, 2024

Vanilla RNNs

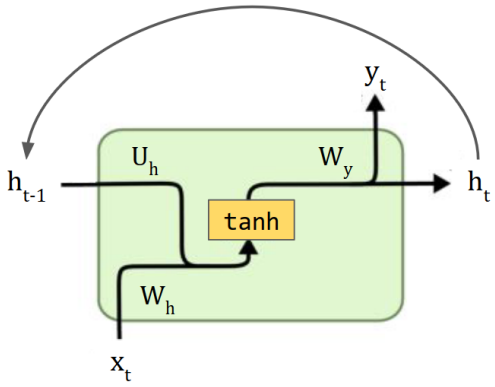


Figure 30: Illustration of a standard RNN layer.

- RNN update equations:
  - $h_t = \sigma_h(W_h x_t + U_h h_{t-1} + b_h)$
  - $y_t = \sigma_y(W_y h_t + b_y)$
    * This part can be any network to perform additional processing to turn hidden state into output
  - x_t is the input, y_t is the output and h_t is the hidden state at time t
  - Each unit contains multiple sets of weights for different purposes
  - σ_h, σ_y are activation functions for the hidden state and output (sigmoid and tanh are 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
  - The input is used to update the hidden state only
- 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
  - For gradient computation, this type of "unrolling" is done
- Since RNNs are unrolled into very long sequences, it is susceptible to a number of problems such as *memory loss*, and vanishing/exploding gradients due to extreme depth

- Exploding gradient can be addressed with gradient clipping, but this damages the gradient
- Using skip connections to address vanishing gradient would require us to keep too many previous states, which is too expensive

RNN Variations

LSTMs and GRUs

- Skip connections to all previous states can be approximated by weighing previous states differently (varying what to forget and what to remember)
- *Gates* can be used to update the context selectively
  - For a matrix $\boldsymbol{X}$, we can control how much of $\boldsymbol{X}$ passes through by multiplying elementwise by some weight matrix
  - The weights are generated from $\boldsymbol{X}$ itself through some activation function, or we can train another neural network to turn $\boldsymbol{X}$ into weights

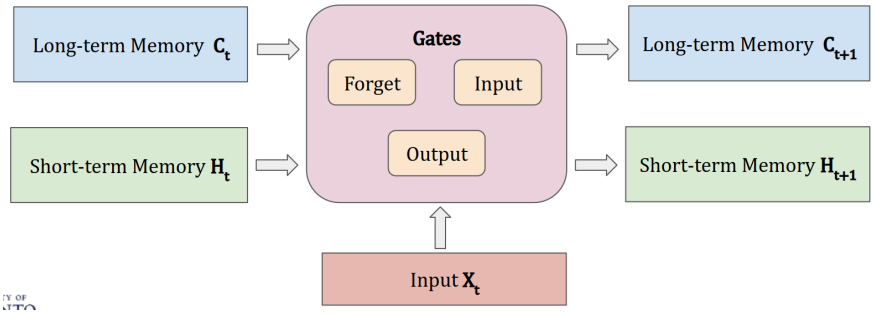


Figure 31: Elements of an LSTM.

- LSTMs (*Long Short-Term Memory*) 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
  - The long-term memory only has things added to it; since we're not multiplying it by a weight matrix each time, this addresses the vanishing and exploding gradient problem
  - There are 4 sets of weights

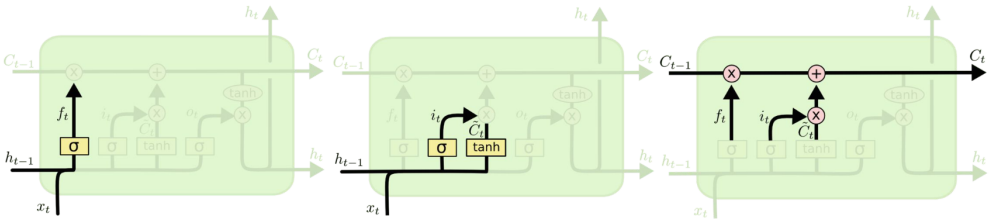
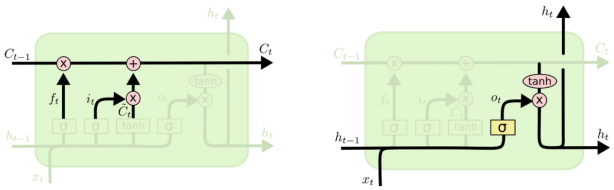


Figure 32: LSTM input processing.

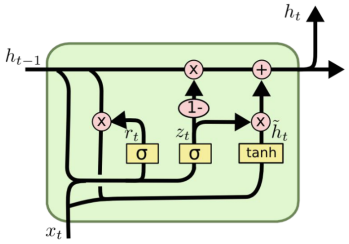
- GRUs combine the forget and input gates into an update gate, and cell state and hidden state are merged



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

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

Figure 33: LSTM output generation.



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

Figure 34: GRU layer architecture.

  - 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
  - Only 3 sets of weights are needed
- In PyTorch, for GRU simply use `nn.GRU` instead of `nn.RNN` as a drop-in replacement
- For LSTM, use `nn.LSTM`; note in the constructor use a tuple of ints to specify both the long and short-term memory sizes, and pass both the short-term and long-term memories to the layer in `forward()`

Deep & Bidirectional RNNs

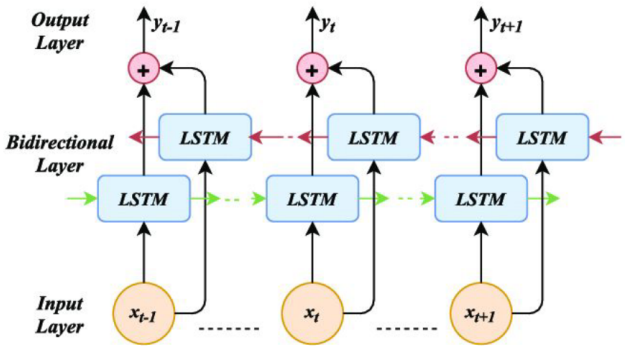


Figure 35: Structure of a bidirectional RNN.

- A typical state in an RNN relies on only the past and present
- 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 or adding the corresponding outputs of the two layers
    * Concatenation typically works better but increases the size

  - Works especially well for applications that require context such as translation
- 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

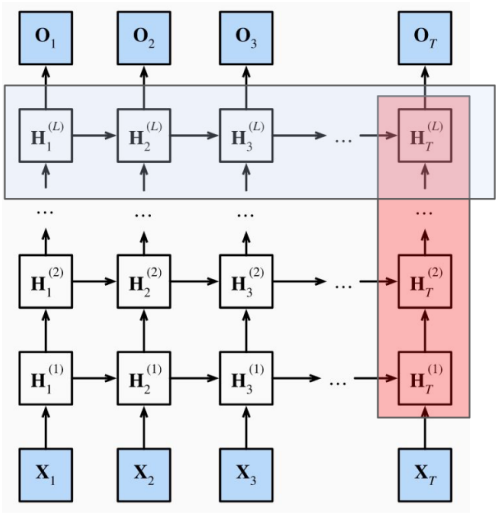


Figure 36: Structure of a deep RNN. The blue highlight is the output; the red highlight is the final hidden state.

- In PyTorch, all the RNN classes mentioned above take `bidirectional` and `num_layers` arguments
  - If `bidirectional` is true, it doubles the output size for some fields
  - To create the initial hidden state for forward passes, we need a tensor with 3 dimensions, the first is the number of layers (doubled for bidirectional), second is batch size, third is hidden size
  - The first element in the tuple of the output has 3 dimensions; the first is the batch size, the second is the input length, the third is the hidden length (doubled for bidirectional)
    * This corresponds to the blue highlight in the figure, and are the final outputs (only consisting of the last layer)
  - The second element in the output tuple has 3 dimensions; the first is the number of layers (doubled for bidirectional), the second is batch size, the third is the output size
    * This corresponds to the red highlight in the figure, and are the final hidden states

Inference Using RNNs

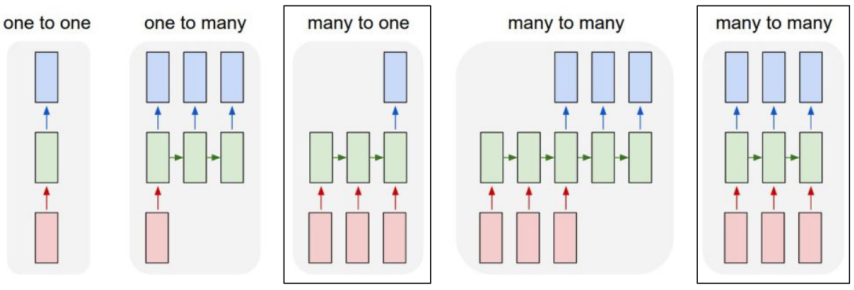


Figure 37: Different types of RNN tasks.

- 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
- The contents of hidden states are different whether we're using an RNN for prediction or generation
  - For prediction we have an encoder-like structure; tokens are processed one at a time, the hidden state represents context of all tokens read so far
  - For generation we have a decoder-like structure; tokens are generated one at a time, and the hidden state represents all the tokens to be generated
- We need to know when to start or stop a generated sequence (since the generated sequence will be of a different length than our input)
  - For this we use dedicated control symbols to denote beginning of sequence and end of sequence (BOS/EOS)
  - Once the model is given BOS, it will start generating
  - Once the model outputs EOS, we know it's done
  - The ground truth output would need to be pre-processed to include the EOS symbol
- 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 the output from the previous step is fed as the new input, and we compare output to the expected second word and so on, until EOS
  - Instead of feeding in the previously generated output, we can instead feed in the ground truth label, so even if the model got it wrong, it still starts the second word with the correct input
    * This is known as *teacher forcing* and helps the model train faster

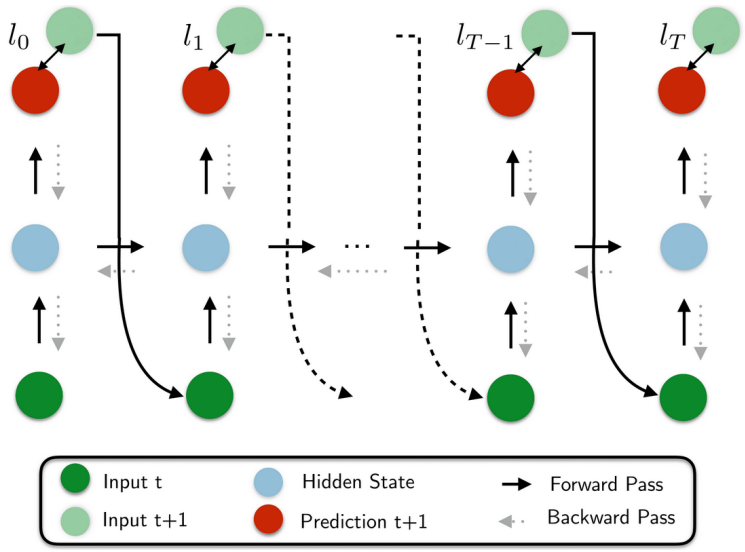


Figure 38: Illustration of teacher forcing.

- At each point in the output sequence, the model gives us an output distribution; how do we pick what word to pick from the distribution?
  - Greedy search: simply select the token with the highest probability at each step
    * Maximize $p(t_1)p(t_2)\cdots p(t_n)$
    * This doesn't work very well and creates many grammatical errors
    * Very cheap to compute since at each step we just need a single output
  - Beam search: search for a sequence of tokens with the highest probability within a window
    * Maximize $p(t_1)p(t_2|t_1)\cdots p(t_n|t_{n-1},\cdots,t_1)$
    * Very expensive because we need to create a tree of possible inputs/outputs, which grows exponentially with window size
    * Within the tree of possible input/output choices, we pick the one that has the highest overall

probability
    * Doing so within all possible words in the vocabulary would be too expensive, so we often only build the tree by branching into the top few words with the highest probabilities

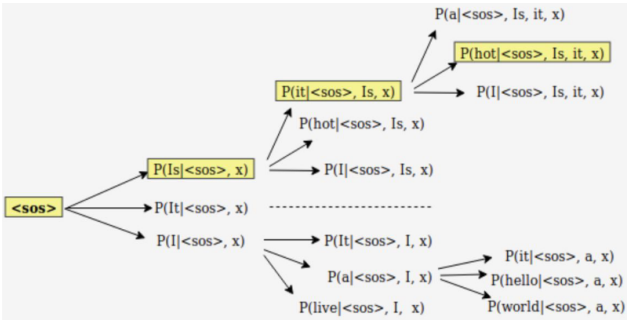


Figure 39: Tree generated by beam search.

- During inference, we don't always want to always select the highest probability output (we don't want output to always be the same), but we cannot have too much diversity or else there would be grammatical errors
    - Softmax temperature scaling: scale the input logits to the softmax by a temperature τ, and then pick the word probabilistically
    - $\frac{e^{z_i/\tau}}{\sum e^{z_i/\tau}}$ where τ is the temperature, a hyperparameter
        * A higher temperature spreads out the distribution
    - At lower temperature, the logits are larger and the model is more confident (distribution is more concentrated)
        * This results in higher quality samples but less variety
    - At higher temperature, the logits are smaller and the model is less confident (distribution is more spread out)
        * This results in lower quality samples with more variety
- For training, we can try to match batches with similar input sizes, so that we reduce the amount of padding (as all inputs are padded to the same length)
    - Use `torchtext.legacy.data.BucketIterator()` with `sort_key=lambda x: len(x.text)`
- The training data will have BOS and EOS added to it, so we remove the BOS from the target and EOS from the input

Lecture 9, Mar 18, 2024

Generative Adversarial Networks (GANs)

- There are a number of different families of deep generative models:
    - Autoregressive models (i.e. RNNs)
    - Variational autoencoders (VAEs)
    - Generative adversarial networks (GANs)
    - Flow-based generative models
    - Diffusion models
- Autoencoders are not great at tasks such as generating faces
    - Autoencoders try to eliminate noise, so they apply a kind of averaging, so the output is often blurry
    - Could we do better by defining a better loss function?
- 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 (often) an unsupervised task since there isn't a ground truth
        * The model learns to approximate $p(x)$, the probability distribution of the samples
        * 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 an encoding of the target (e.g. one-hot encoding of the 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 a training dataset) or fake
        * This is a supervised task since the output is a real/fake label
            - The output is a binary true/false so the output layer is always a single neuron
        * The model learns to approximate $p(y|x)$, the probability that a sample is real given the sample
        * 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 realistic samples, while the discriminator learns the difference between real and fake samples
    - The models are trained at the same time and learn off of each other
    - Since the goal of the generator is to fool the discriminator, its loss is defined based on the discriminator

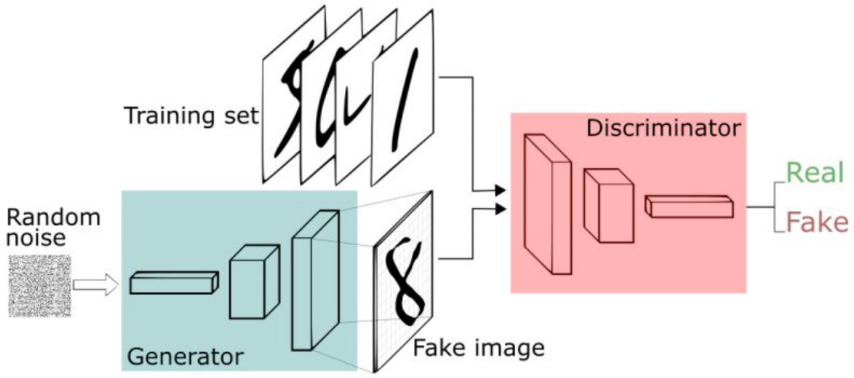


Figure 40: Structure of a GAN.

GAN Training

- The discriminator loss is simply binary cross-entropy; for a real image, we expect a real output, while for a generated image we expect a fake output
- The generator loss is also binary cross entropy, but on the discriminator output
    - The discriminator will be fed a generator output; as the generator, we expect the output to be real, since we're trying to fool the discriminator
    - The gradient is backpropagated through the discriminator first, and then the generator
    - The discriminator weights are frozen while the generator is being trained
- To train, alternative between training the discriminator and generator
    - First train the discriminator for k times:
        * Sample m noise samples, pass them through the generator to obtain generated samples, and then input to the discriminator
            - For these, the ground truth label is false
        * Sample m samples from the training dataset, input into the discriminator
            - For these, ground truth is true
        * Use BCE loss function to compute gradients
    - Then train the generator for one time:

        * Sample m noise samples, pass through generator and discriminator
        * Use BCE to compute gradients, with ground truth label being true (real)
        * Update generator weights according to the gradient, but do not update the discriminator (since this would make it worse)
    - k is a hyperparameter used to balance the training between generator and 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)
    - Mode collapse can transfer to a different mode
- Since there are 2 competing processes, GANs are very hard to train and take very long
    - 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

Architecture Variations

- To control the class that we generate, we can use a conditional GAN
    - The generator is passed the noise and the condition, which can be e.g. a one-hot encoding of the category
    - The discriminator is passed the same condition

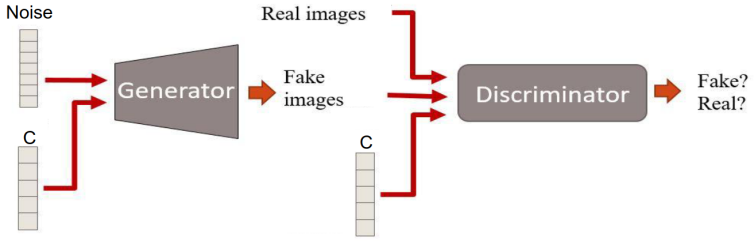


Figure 41: Conditional generation.

- *Style transfer* is a technique using GANs to transform an image into a different domain
    - The generator input is an image that we want to do style transfer on
    - Using a conventional GAN architecture without the feedback cycle, the generator will simply treat the input as random noise and generate an image in the style desired, but without characteristics of the original input
- CycleGAN is an architecture that can be used for style transfer
    - Introduce an additional model, which transforms the images back into the original domain
    - Reconstruction loss between the resulting image and the original image is added to the loss
    - This is similar to an autoencoder; the image with style transferred is the embedding

Adversarial Attacks

- Goal: choose a small perturbation ϵ on an image x such that a neural network f misclassifies $x + \epsilon$
- These attacks can often be very effective; the amount of noise added is often unnoticeable to a human, but completely changes the model's output
- To do this we optimize ϵ as a parameter and apply backpropagation; 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

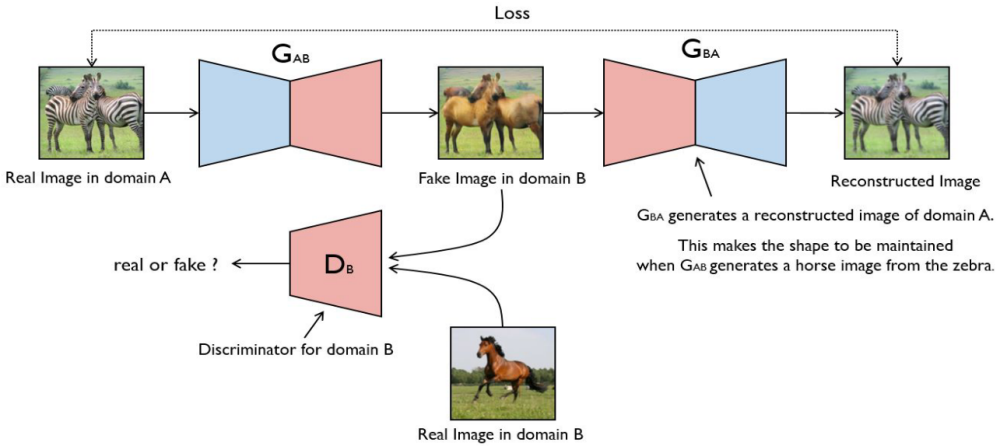


Figure 42: CycleGAN architecture for style transfer.

  - 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

Lecture 10, Mar 25, 2024

Attention Mechanisms

- The main building block of transformers
- In an attention mechanism, the model learns an *attention score* for different parts of the input sequence
  - Parts with higher attention score are deemed more important, e.g. adjectives on nouns, compared to connecting words
- The attention scores are used to aggregate the data, e.g. by taking a weighted sum, before it is passed to the rest of the model
- Example: attention-based 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, but relative 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
    * Note that this simple attention mechanism does not track order
- Attention can be defined between two elements, which gives 2 types:
  - *Cross-attention*: between two sequences
    * e.g. in translation, we compute the attention between words in two languages
  - *Self-attention* (or *intra-attention*): input with respect to itself, i.e. for a token, compute the attention for all other tokens in the same sequence
    * This can be used to match different words of the input, e.g. figuring out what the word "it" refers to

- Given two embeddings we have many different methods to compute their attention scores
  - Methods such as dot product or cosine similarity are simple but has no weights to learn
  - We can use a bilinear form $a^T W b$ to add weights
  - Can also use more complex forms such as a fully-connected network

Transformer Networks

- RNNs' sequential nature means we can't parallelize them easily, making them slow to train
- Transformers are a type of network based solely on self-attention
- Attention is modelled as a "soft 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 $X \in \mathbb{R}^{n \times i}$, queries $Q \in \mathbb{R}^{n \times k}$, keys $K \in \mathbb{R}^{n \times k}$, values $V \in \mathbb{R}^{n \times v}$, are generated from the same input using 3 different linear layers with different weights
  - This is equivalent to a matrix multiplication
  - The queries and keys have the same dimension, but the values do not
    * Both dimension sizes are hyperparameters
  - 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*
  - QK^T would give us the pairwise attention scores between all tokens within the input sequence
  - d_k is the hidden dimension of Q and K (i.e. the dimensionality of the embeddings); this keeps the scores from increasing artificially with embedding size
  - After a softmax, we get the normalized attention scores ($n \times n$), which we then multiply by V
- In the end, each token in the result is a weighted sum of all other tokens, weighted by how important the other tokens are to that token
  - We now have a new embedding that integrates the surrounding context
  - Unlike word2vec or GloVe these are context-sensitive, since the meaning of a word depends on its surroundings
  - This process is easily parallelized since it is not sequential, and doesn't suffer from memory loss

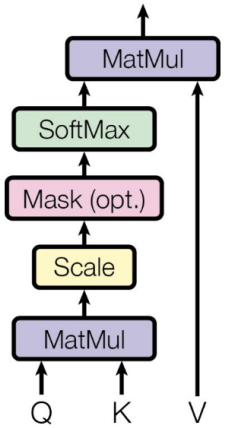


Figure 43: Scaled dot-product attention.

- *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
  - Before passing to each attention head and after concatenation, we use linear layers
  - Instead of having one very strong model, we have several weaker models that work together
  - This tends to increase performance

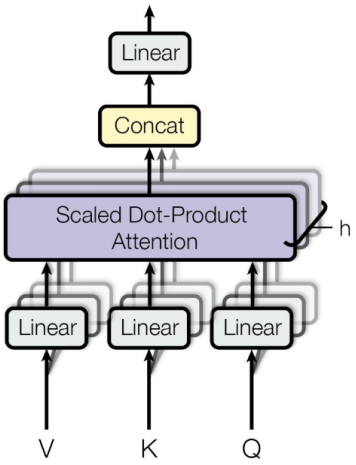


Figure 44: Multi-head attention mechanism.

- Each transformer encoder layer consists of a multi-head self-attention sublayer, followed by a fully connected sublayer (which can be deep), with residual (skip) connections around each of the sublayers followed by layer normalization
  - The output of the multi-head attention is normalized (layer norm)
  - We can stack transformer layers just as with any other type of network
- The output from the transformer encoder is an embedding that we can feed to further layers for processing

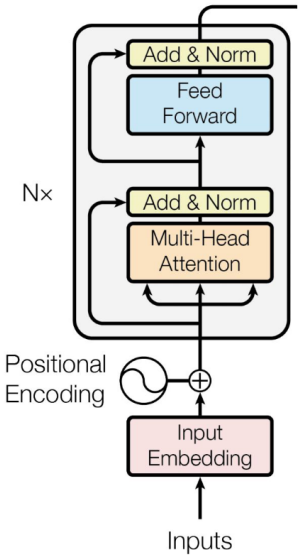


Figure 45: Structure of a transformer layer.

- After the transformer, we pool its output using methods such as summing, which produces embedding sizes that do not depend on the input length

  - Since the model has no recurrent or convolutional layers, it doesn't take into account order by itself
- We need positional encoding to give the model order information
  - One way to do this is alternate between sines and cosines
  - The goal is to create an encoding in the same shape as the input word embeddings, with a unique value for each position in the sequence and embedding
  - This is then added elementwise to the input embeddings before being passed into the encoder layer
  - The positional encoding is defined outside the model
- Transformers have a number of advantages over RNNs:
  - Good with long-range dependencies (no memory loss since everything has a direct path to output, regardless of its position in the input sequence)
  - 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)
- In PyTorch, use linear layers to compute the Q, K, V matrices, and use `nn.MultiheadAttention(hidden_size, num_heads, batch_first=True)` for the attention
  - `nn.MultiheadAttention` is called with Q, K, V matrices as arguments

Applications

- Most common application is in language processing since it can process context
- Example: Google's BERT (Bidirectional Encoder Representations from Transformers) model
  - Trained on two self-supervised tasks:
    * Masked word prediction: randomly mask 15% of tokens and predict what was masked
      - Similar to a de-noising encoder
      - Loss is computed on masked words only (otherwise the model memorizes the unmasked words)
    * Next sentence prediction: predicting if two sentences are likely to appear together (binary output)
  - Inputs are the elementwise sum of embeddings for each token, embeddings for each sentence (same embeddings for all the words in a sentence) and positional encodings
    * Uses special control tokens to determine the task and sentence separation, etc
  - The transformer part can be isolated and used for transfer learning, like 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
  - Each patch is flattened and passed through fully connected layers to first obtain an embedding
  - If the patches are small enough, we lose no spacial information
  - The transformer itself does not change

Lecture 11, Apr 1, 2024

Graph Neural Networks (GNNs)

- We might want to give inputs to our models that might have general, non-Euclidean relationships, such as a molecule, or a 3D structure, etc
- Architectures we've seen so far such as CNNs or RNNs cannot handle generic relationships such as these
- In general, to achieve order invariance we first learn embeddings for each item using a shared network

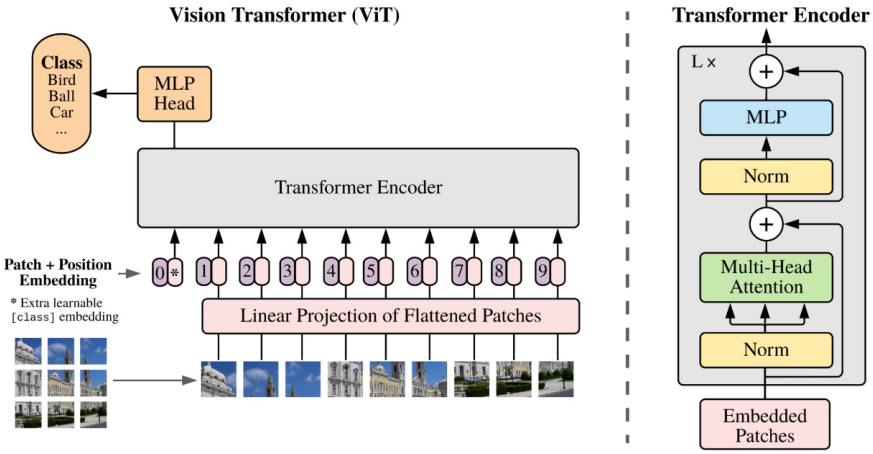


Figure 46: Structure of a vision transformer.

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*
  - Similar to putting words into GloVe embeddings first and then summing, except in this case we want order invariance
- 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*)
  - Since the transformer learns an attention matrix, it creates a fully-connected graph over all nodes in the input and learns the edge weights
- 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 $a_{ij} = \begin{cases} 1 & (i,j) \in E \\ 0 & (i,j) \notin E \end{cases}$

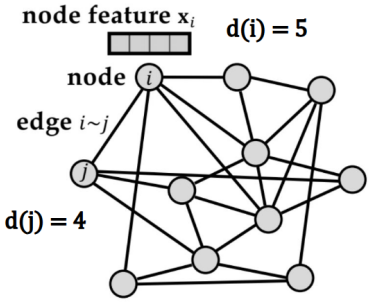


Figure 47: Example graph.

- 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
  - Feeding the adjacency matrix to a network directly doesn't work since there are $n!$ distinct adjacency matrices that describe the same graph, so the network needs to learn all of them
- We want our model, which is a function on a graph, to have 2 main properties:
  - *Invariance*: output does not change in response to changes in input ordering
    - e.g. if we input a molecule, the order that we put the atoms in should not affect whether the model predicts it is toxic
  - *Equivariance*: output (at the node level) properly changes in response to changes in input ordering
    - e.g. if we change the order of the atoms, then the order of the classes of atoms changes with it appropriately
- 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 adjacency matrix A 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, e.g. what kind of atom is each node), graph classification (operating on the entire graph, e.g. what kind of molecule is the entire graph), or link prediction (operating on edges, e.g. what kind of bond is between two atoms)
- 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
  - $h_v^{(k)} = \text{COMBINE}^{(k)}\left(h_v^{(k-1)}, \text{AGGREGATE}^{(k)}\left(\left\{ \left(h_v^{(k-1)}, h_u^{(k-1)}, e_{uv}\right) \,\middle|\, u \in \mathcal{N}(v) \right\}\right)\right)$
    - $\mathcal{N}(v)$ are the neighbours of node v
    - AGGREGATE is an order-invariant function, but COMBINE may not be (we can use operations such as concatenate, or another neural network)
- With a single message passing update, each node receives the embeddings from all its immediate neighbours; but with more updates, information from further nodes can propagate and reach the node as well
  - The more updates we do, the larger the receptive field of each node
- To get data out of the graph, we use a *read-out* (pooling) function, which must be another order-invariant function
  - $h_G = \text{READOUT}(\{ h_v^{(K)} \mid v \in G \})$
  - This gives us a graph embedding which we can pass to another network

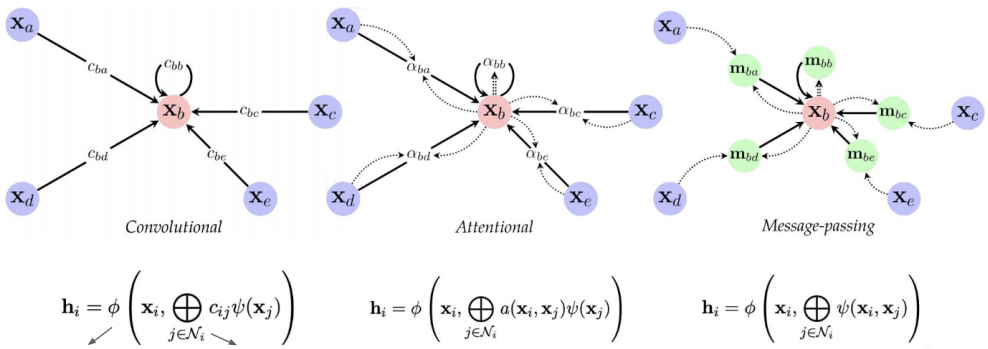


Figure 48: Different choices of aggregation and combine functions create different types of GNNs.

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
  - Both A and X are given inputs; we learn the weights W
- 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. $A \leftarrow A + I$
    * This essentially does aggregate and combine at the same time
  - 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 $H = \sigma\left(D^{-\frac{1}{2}}AD^{-\frac{1}{2}}XW\right)$ (assuming A has already been aggregated by identity)
- A GCN layer will only update the node embeddings based on the immediate ("1-hop") neighbours, since A will only be nonzero where there is an edge
  - To get influence from further nodes, we have to apply the GNC update multiple times
    * Each time we update, we use a different weight matrix
    * This is analogous to adding deeper layers
  - This is analogous to increasing the receptive field in CNNs, since as we do more updates, further neighbours can start influencing each node

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

GNN Implementation

- In a naive dense implementation, we simply use `nn.Linear()` to make the weight matrix W, and perform the matrix operations normally with `torch.mm()`
  - We add identity to the input matrix and apply the normalization to compute $D^{-\frac{1}{2}}(A + I)D^{-\frac{1}{2}}$
    * To get the degree matrix D we take a sum across each row and create a diagonal matrix, `torch.diag(torch.sum(A, dim=0))`, since this counts the number of connections for each node
  - Use the linear layer to compute $D^{-\frac{1}{2}}(A + I)D^{-\frac{1}{2}}XW$, and then apply the activation function to finish the layer
  - Get the graph embedding by pooling all the resulting embeddings through, e.g. a sum with `torch.sum(x, dim=0)`
  - Pass the graph embedding through more (normal) fully connected layers to get the final prediction
- However, most graphs are sparse, so using the adjacency matrix A is unnecessarily expensive
- The PyTorch Geometric (PYG) library allows us to use a sparse representation that is more efficient
- Using `torch_geometric.data.Data(x, edge_index, y)` we can represent graphs
  - `x` is an $N \times D$ tensor that contains all the feature vectors
  - `edge_index` is a $2 \times M$ tensor where matching indices in the two rows denote edges
  - `y` is the ground truth label tensor
- Using `torch_geometric.nn.GCNConv(in_features, out_features)` we can replace all the matrix operations by just calling the layer as a function

  - Normalization and adding identity is taken care of by `GCNConv`
  - The layer expects `x` and `edge_index` as the inputs, which we get from the `Data` object
  - We still need to apply the activation function
- `torch_geometric.nn.GATConv(in_features, out_features)` can be used in place of `GCNConv()` as well
- For a dense implementation, the batched adjacency matrix would be done by creating a diagonal matrix of adjacency matrices
  - This would make it even less efficient since the matrix would be filled with zeroes
- In the sparse implementation uses an index vector that maps each node to its respective graph in the batch
  - e.g. if we have $\begin{bmatrix}0 & 0 & 0 & 1 & 1 & 1\end{bmatrix}^T$ this means the first 3 nodes belong to graph 0, while the last 3 nodes belong to graph 1
  - Use a `torch_geometric.data.DataBatch(batch, edge_index, x, y)` where `batch` is the batch index vector
  - `torch_geometric.loader.DataLoader(dataset, batch_size, shuffle)` is the data loader type for `torch_geometric`
    * `torch_geometric.datasets` has common datasets that can be loaded, similar to torchvision
  - Use `torch_geometric.nn.global_add_pool(x, batch)` to do add pooling with proper batching (since we can no longer do a simple `torch.sum()`)
    * The result is still an embedding tensor that can be passed to FC layers as usual
- Training works in the exact same way as any other model