18-661 Introduction to Machine Learning

Neural Networks-III

Spring 2025

ECE – Carnegie Mellon University

Announcements

- On wednesday, March 26th, we will have Mini-exam 2 during the first half of the lecture. You are allowed to bring 1 single-sided handwritten cheatsheet.
- Recitation on Friday will go over some practice problems for the mini-exam next week
- Homework 3 is due on Friday March 28th.
- On Monday, we will have a PyTorch tutorial lecture.

Outline

1. Review: Training a Neural Network: Backpropagatiopn
2. Review: Optimizing SGD Parameters for Faster Convergence
3. Mitigating Overfitting
4. Language Models and RNNs
5. Transformer Language Models

Review: Training a Neural Network: Backpropagatiopn

Learning Parameters

How to learn the parameters?

- Choose the right loss function
  - Regression: Least-square loss (last week's lectures)

$$min \sum_n (f(\mathbf{x}_n) - t_n)^2$$

  - Classification: cross-entropy loss (in the homework)

$$min - \sum_n \sum_k t_{nk} \log f_k(\mathbf{x}_n) + (1 - t_{nk}) \log(1 - f_k(\mathbf{x}_n))$$

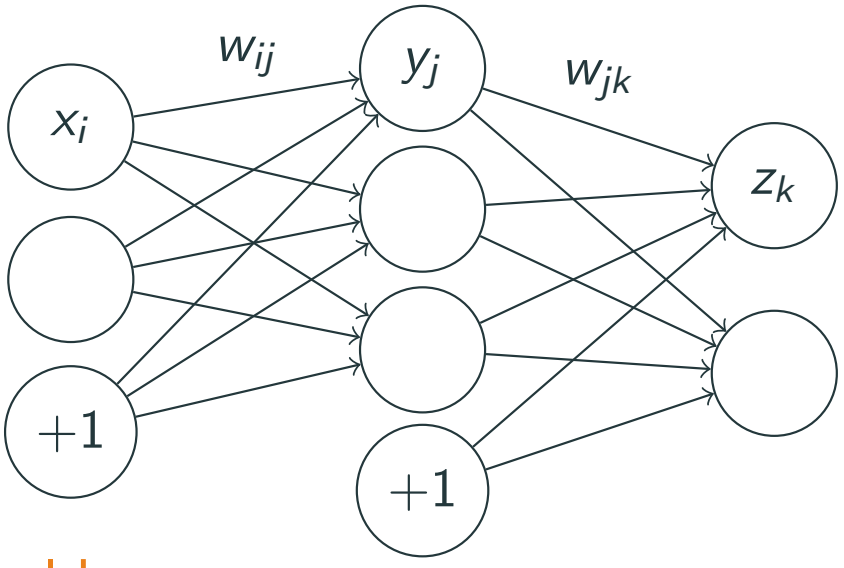
- Hard optimization problem because f (the output of the neural network) is (usually) a complicated function of $\mathbf{x}_n$
  - Stochastic gradient descent is commonly used
  - Many optimization tricks are applied

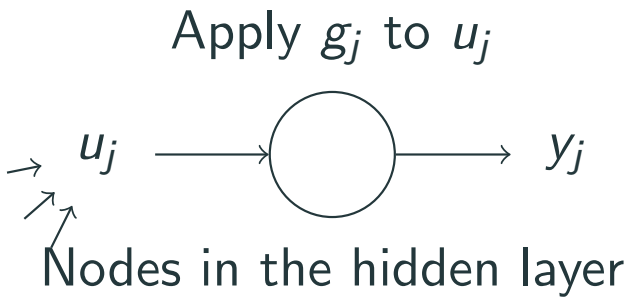
Updating the Parameter Values

Back-propagate the error. Given parameters w, b:

- Step 1: Forward-propagate to find the output z_k in terms of the input (the "feed-forward signals").
- Step 2: Calculate output error E by comparing the predicted output z_k to its true value t_k.
- Step 3: Back-propagate E by weighting it by the gradients of the associated activation functions and the weights in previous layers.
- Step 4: Calculate the gradients $\frac{\partial E}{\partial w}$ and $\frac{\partial E}{\partial b}$ for the parameters w, b at each layer based on the backpropagated error signal and the feedforward signals from the inputs.
- Step 5: Update the parameters using the calculated gradients $w \leftarrow w - \eta \frac{\partial E}{\partial w}$, $b \leftarrow b - \eta \frac{\partial E}{\partial b}$ where η is the step size.

High-level Procedure: Can be Used with More Hidden Layers





Final Layer

- Error in each of its outputs is $\frac{\partial E}{\partial z_k} = z_k - t_k$.
- Error in input u_k to the final layer is $\delta_k = \frac{\partial E}{\partial u_k} = g_k'(u_k)(z_k - t_k)$

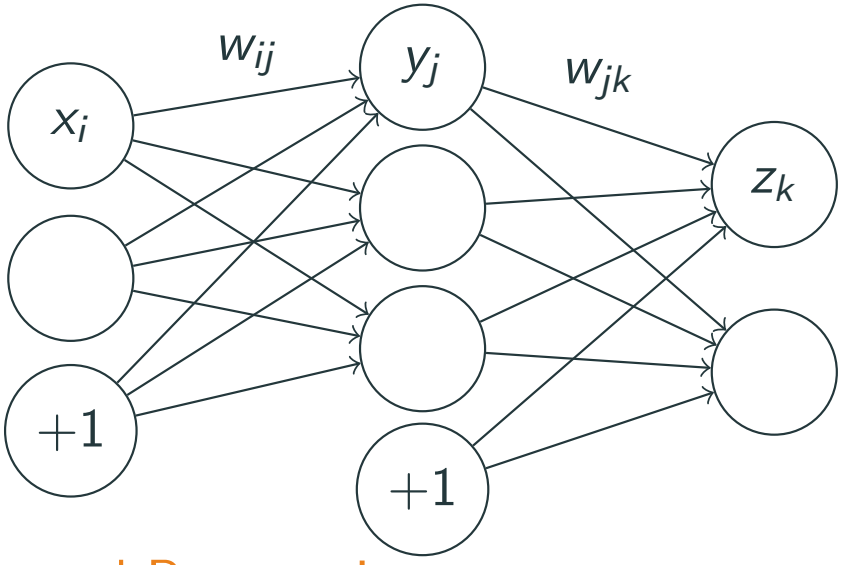
Hidden Layer

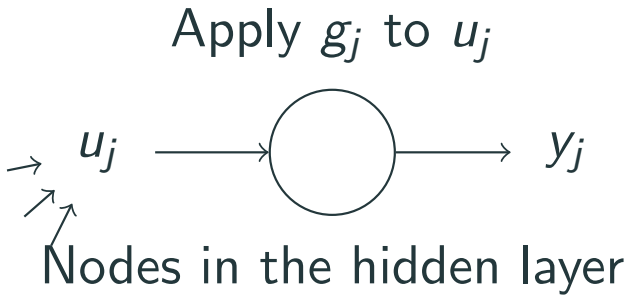
- Error in output y_j is $\sum_{k \in K} \delta_k w_{jk}$.
- Error in the input u_j is $\delta_j = \frac{\partial E}{\partial u_j} = g_j'(u_j) \sum_{k \in K} \delta_k w_{jk}$

The gradients w.r.t. w_{jk} and w_{ij} are $y_j \delta_k$ and $x_i \delta_j$.

Vectorized Implementation

Much faster than implementing a loop over all neurons in each layer



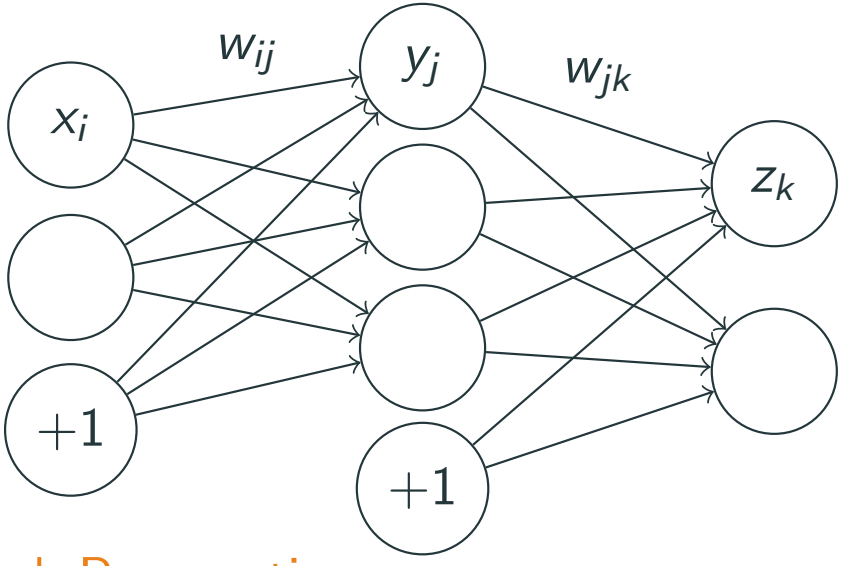


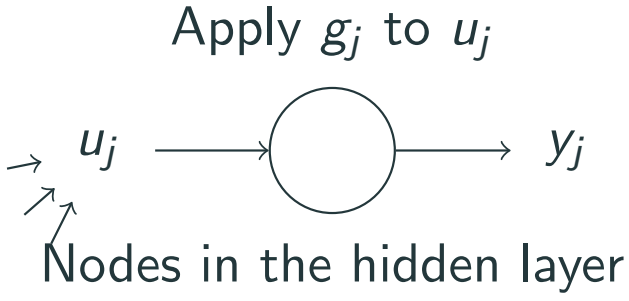
Forward-Propagation

- Represent the weights between layers $\ell - 1$ and ℓ as a matrix $\mathbf{W}^{(\ell)}$, where the (j, k) entry in $\mathbf{W}^{(\ell)}$ is w_{jk}.
- Outputs of layer $\ell - 1$ are in a row vector $\mathbf{y}^{(\ell-1)}$. Then we have $\mathbf{u}^{(\ell)} = \mathbf{y}^{(\ell-1)}\mathbf{W}^{(\ell)}$.
- Outputs of layer ℓ are in the row vector $\mathbf{y}^{(\ell)} = g(\mathbf{u}^{(\ell)})$.

Vectorized Implementation

Much faster than implementing a loop over all neurons in each layer





Back-Propagation

- For each layer l find $\Delta^{(\ell)}$, the vector of errors in $\mathbf{u}^{(\ell)}$ in terms of the final error E.
- Recursively find $\Delta^{(\ell-1)}$ in terms $\Delta^{(\ell)}$: $\Delta^{(\ell-1)} = g'\left(\mathbf{u}^{(\ell-1)}\right) \cdot \left(\mathbf{W}^{(\ell)}\Delta^{(\ell)}\right)^T$, where $\cdot$ denotes element-wise multiplication.
- Update weights $\mathbf{W}^{(\ell)}$ using $\Delta^{(\ell)}$

Basic Idea behind DNNs

Architecturally, a big neural network (with a lot of variants)

- in depth: 4-5 layers are common (Google LeNet uses more than 20)
- in width: each layer might have a few thousand hidden units
- the number of parameters: hundreds of millions, even billions

Algorithmically, many new things, including:

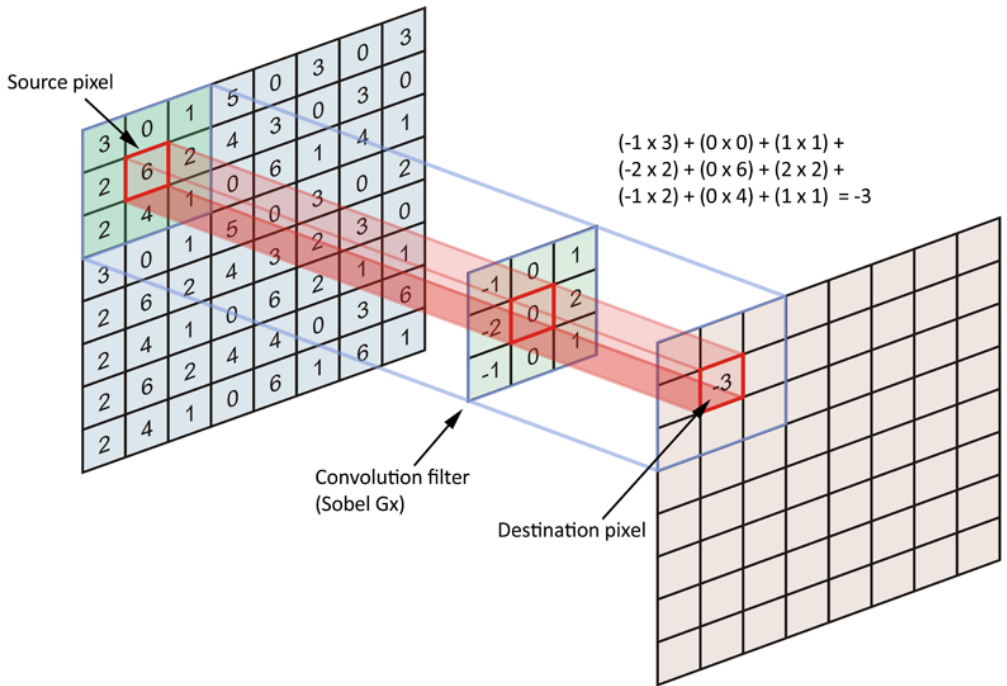
- Pre-training: do not do error backpropagation right away
- Layer-wise greedy: train one layer at a time

Computing

- Requires fast computations and coping with a lot of data
- Ex: fast Graphics Processing Unit (GPUs) are almost indispensable

Deep Convolutional Networks

- Compared to standard feedforward neural networks with similarly-sized layer
  - CNNs have much fewer connections and parameters
  - and so they are easier to train
  - while their theoretically-best performance is likely to be only slightly worse.
- Usually applied to image datasets (where convolutions have a long history).



Review: Optimizing SGD Parameters for Faster Convergence

(Full-)Batch vs. Stochastic GD

- Recall the empirical risk loss function that we considered for the backpropagation discussion

$$E = \sum_{n=1}^{N} \frac{1}{2}(f(\mathbf{x}_n) - t_n)^2$$

- For large training datasets (large N), then computing gradients with respect to each datapoint is expensive. For example, for the last layer, the batch gradients are

$$\frac{\partial E}{\partial w_{jk}} = \sum_{n=1}^{N}(z_{k,n} - t_{k,n})$$

- Therefore we use stochastic gradient descent (SGD), where we choose a random data point $\mathbf{x}_n$ and use $E = \frac{1}{2}(f(\mathbf{x}_n) - t_n)^2$ instead of the entire sum

Mini-Batch SGD

- Mini-batch SGD is in between these two extremes
- In each iteration, we choose a set S of m samples from the N training samples and use

$$E = \sum_{n \in S} \frac{1}{2}(f(\mathbf{x}_n) - t_n)^2$$

  for backpropagation
- Small m saves per-iteration computing cost, but increases noise in the gradients and yields worse error convergence
- Large m reduces gradient noise and gives better error convergence, but increases computing cost per iteration

How to Choose Mini-Batch Size

- Small training datasets – use batch gradient descent $m = N$
- Large training datasets – typical m are 64, 128, 256 ... whatever fits in the CPU/GPU memory
- Mini-batch size is another hyperparameter that you have to tune

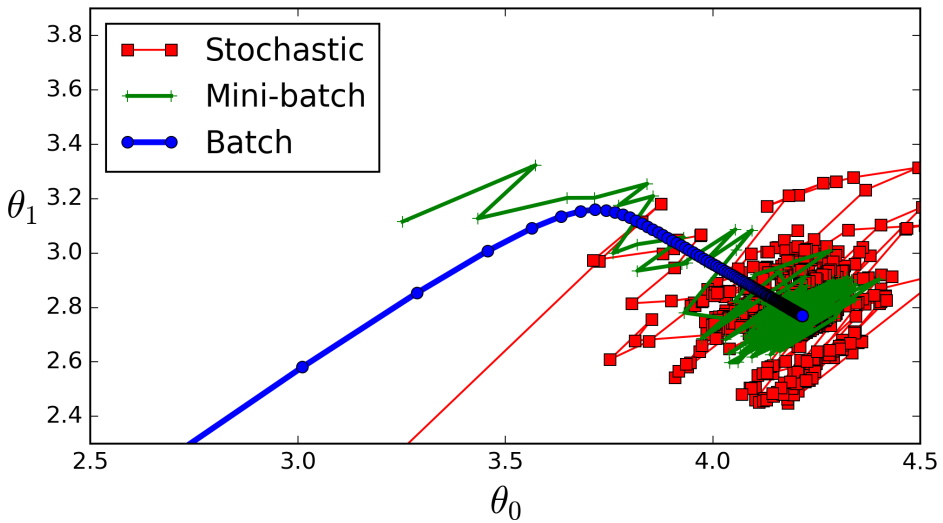


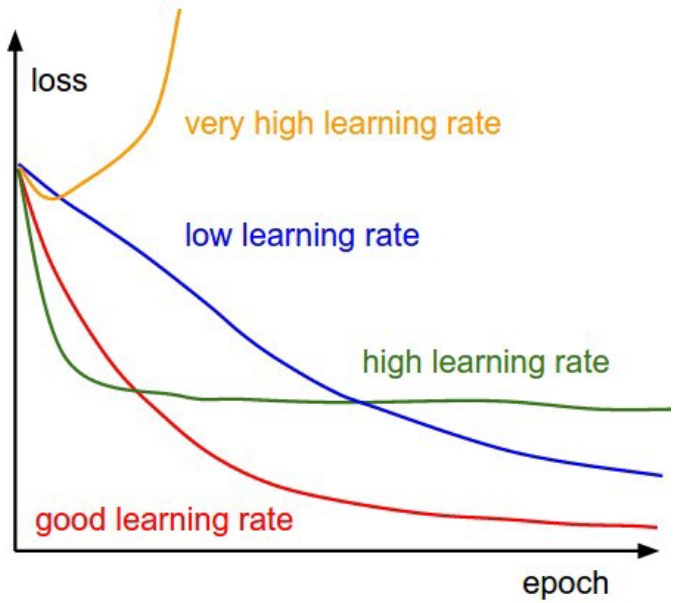
Image source: `https://github.com/buomsoo-kim/Machine-learning-toolkits-with-python`

Learning Rate

- SGD Update Rule

$$w^{(t+1)} = w^{(t)} - \eta \frac{\partial E}{\partial w^{(t)}} = w^{(t)} - \eta \nabla E(w^{(t)})$$

- Large η; Faster convergence, but higher error floor (the flat portion of each curve)
- Small η: Slow convergence, but lower error floor (the blue curve will eventually go below the red curve)
- To get the best of both worlds, decay η over time



How to Decay the Learning Rate?

A common way to decay η

- Start with some learning rate, say $\eta = 0.1$
- Monitor the training loss and wait till it flattens
- Reduce η by a fixed factor, say 5. New $\eta = 0.02$.
- Reduce again by the same factor when curve flattens

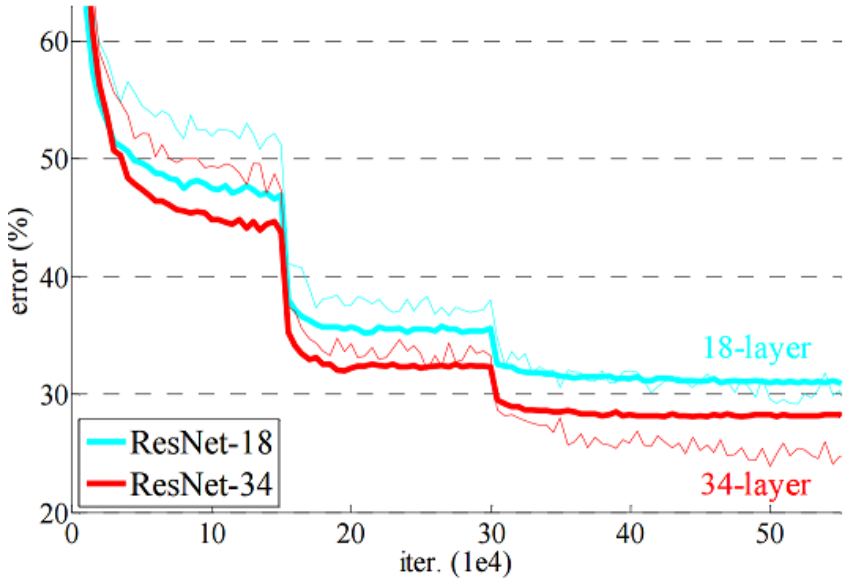


Image Source: `http://www.bdhammel.com/learning-rates/`

How to Decay the Learning Rate?

An alternate approach – AdaGrad [Duchi et al 2011]

- Divide the learning rate η by the square root of the the sum of squares of gradients until that time
- This scaling factor is different for each parameter depending upon the corresponding gradients

$$\mathbf{w}_i^{(t+1)} = w^{(t)} - \frac{\eta}{\sqrt{g_i^{(t)} + \epsilon}} \nabla E(\mathbf{w}_i^{(t)})$$

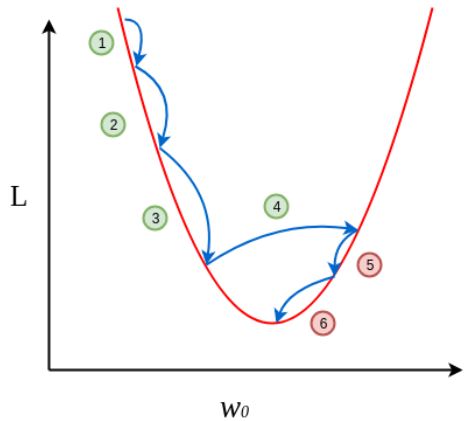
  where $g_i^{(t)} = \sum_{k=1}^{t} (\nabla E(\mathbf{w}_i^{(k)}))^2$.
- In a modified version AdaDelta, you take the sum of square gradients over a fixed size sliding window instead of all times from 1 to t

Momentum

- Momentum SGD: Accelerates SGD by considering the inertia of motion to speed-up future steps.
- Formal update rule is given by:

$$v^{(t)} = \gamma v^{(t-1)} + \eta \nabla E(w^{(t)})$$
$$w^{(t+1)} = w^{(t)} - v^{(t)}$$

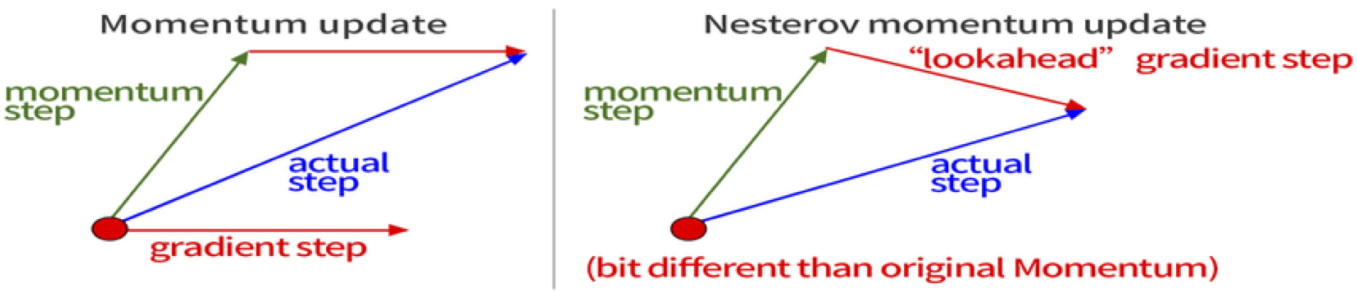
  where the update vector $v^{(t)}$ in the $t+1$-th iteration includes the gradient plus a small fraction γ of the previous update vector
- The momentum γ is usually set to 0.9



Nesterov Momentum

- Momentum can speed-up the ball too much and cause it to overshoot the minimum
- Nesterov momentum modifies the update rule slightly so as to anticipate future changes in the gradient

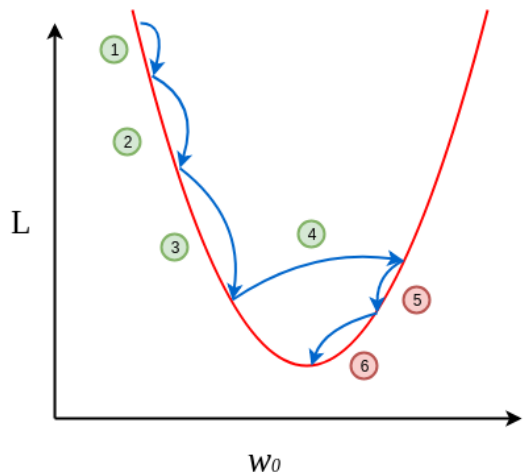
$$v^{(t)} = \gamma v^{(t-1)} + \eta \nabla E(w^{(t)} - \gamma v^{(t-1)})$$
$$w^{(t+1)} = w^{(t)} - v^{(t)}$$



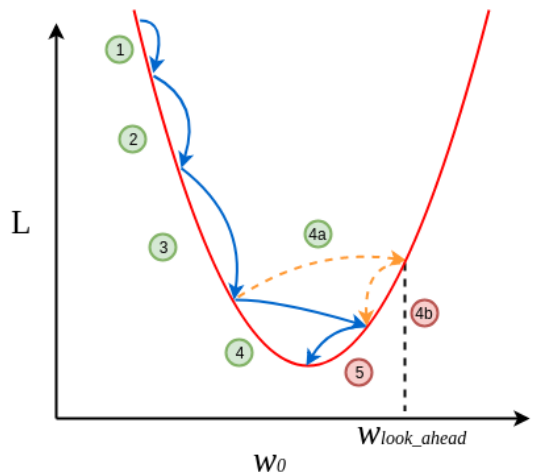
Nesterov Momentum

- Momentum can speed-up the ball too much and cause it to overshoot the minimum
- Nesterov momentum modifies the update rule slightly so as to anticipate future changes in the gradient

$$v^{(t)} = \gamma v^{(t-1)} + \eta \nabla E(w^{(t)} - \gamma v^{(t-1)})$$
$$w^{(t+1)} = w^{(t)} - v^{(t)}$$



(a) Momentum-Based Gradient Descent



(b) Nesterov Accelerated Gradient Descent

$\bigcirc \Longrightarrow \frac{\partial L}{\partial w_0} = \frac{Negative(-)}{Positive(+)}$ $\bigcirc \Longrightarrow \frac{\partial L}{\partial w_0} = \frac{Negative(-)}{Negative(-)}$

Adam

- Adaptive Moment Estimation (Adam) is a method that first appeared in [Kingma, Ba 2015] – it combines the adaptive learning rate idea used in AdaDelta with an adaptive momentum term

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \quad \text{First moment gradient estimate} \quad (1)$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \quad \text{Second moment gradient estimate} \quad (2)$$

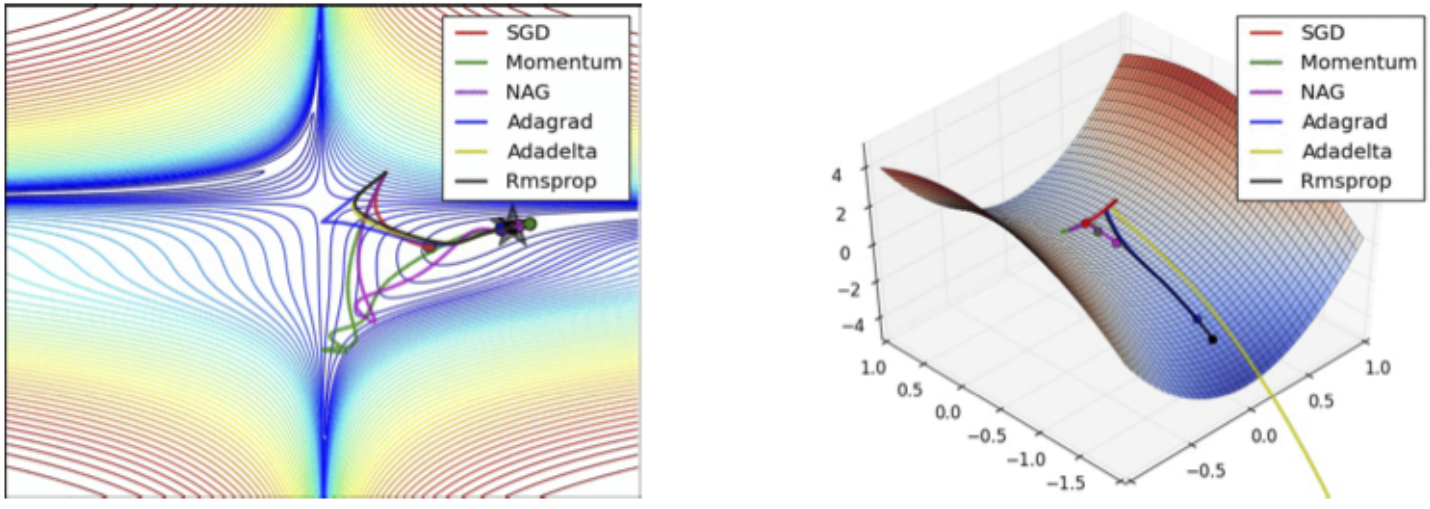
- Model parameters are updated as:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \left(\frac{\eta}{\sqrt{v_t/(1-\beta_2^t) + \epsilon}} \right) \frac{m_t}{1-\beta_1^t}$$

- Other adaptive methods include RMSprop, AdaMax, Nadam etc.

Which optimizer should one use?

- For sparse parameter vectors, adaptive optimizers work very well, and Adam typically outperforms other adaptive methods. However, the exact trend varies with the dataset and loss function



(a) SGD optimization on loss surface contours

(b) SGD optimization on saddle point

Figure 4: Source and full animations: Alec Radford

Full animation can be found here: `http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html`

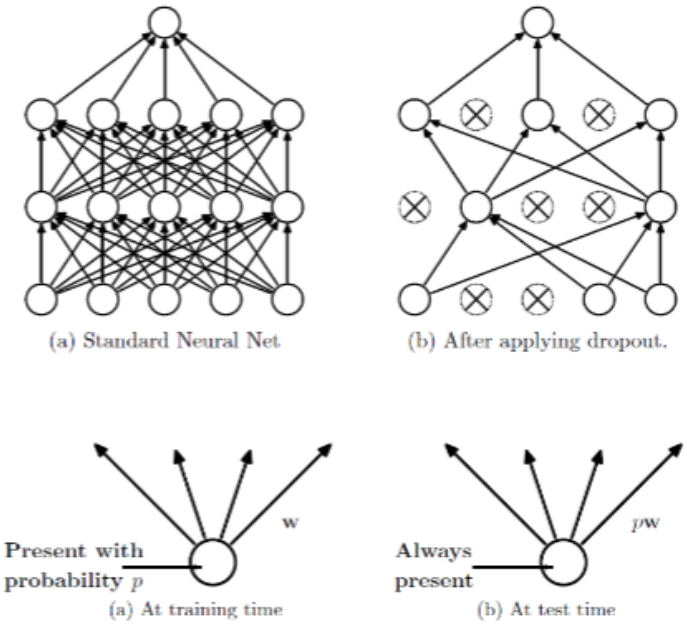
Mitigating Overfitting

Preventing Overfitting

- Approach 1 Get more data
  - Always best if possible!
  - If no natural ones, use data augmentation
- Approach 2 Regularization
  - Add regularizer term to the objective function
  - Need to incorporate this in back-propagation
- Approach 3 Choose network structure with the right capacity:
  - Enough to fit the true data variations, but not to fit noise
  - Requires parameter tuning, hard to guess the right size.
- Approach 4 Average many different models
  - Models with different forms to encourage diversity (Dropout)
  - Train on different subsets of data
- Approach 5 Batch Normalization
  - Scaling and shifting of features to reduce overfitting

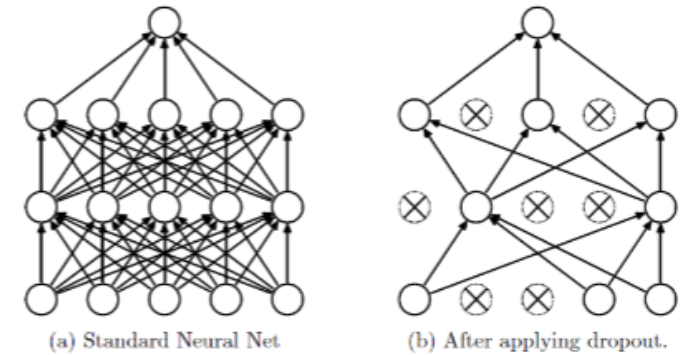
Dropout

- Consider a fully connected neural net with H nodes in hidden layers.
- Each time we present a training example (for each iteration of SGD), we randomly omit each hidden unit with probability 0.5.
- So we are randomly sampling from 2^H different architectures.
- All architectures share weights.



Dropout as Preventing Co-adaptation

- If a hidden unit knows which other hidden units are present, it can co-adapt to them on the training data.
  - But complex co-adaptations are likely to go wrong on new test data.
  - Big, complex conspiracies are not robust.



Dropout as a Form of Model Averaging

- We sample from 2^H models. So only a few of the models ever get trained, and they only get one training example.
- The sharing of the weights means that every model is very strongly regularized.
  - It's a much better regularizer than L2 or L1 penalties that pull the weights towards zero.
  - Note that it's hard to generalize dropout to other types of ML models, unlike L2 or L1 penalties.

What Do We Do at Test Time?

- We could sample many different architectures and take the mean of their output distributions.
- It's better to use all of the hidden units, but to halve their outgoing weights.
  - This exactly computes the geometric mean of the predictions of all 2^H models! (see paper on Understanding dropout)
  - This is not exactly the same as averaging all the separate dropped out models, but it's a pretty good approximation, and it's fast.
- Alternatively, run the stochastic model (i.e., the different architectures) several times on the same input.
  - This gives us an idea of the uncertainty in the answer.

Some Dropout Tips

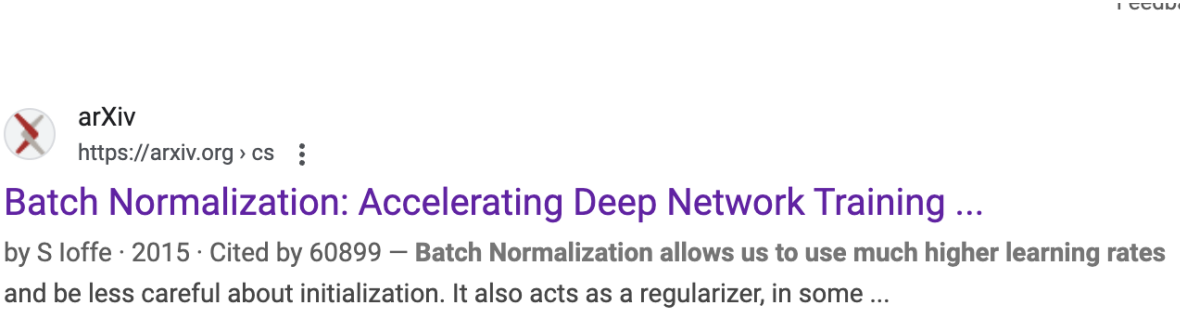
- Dropout lowers your **capacity**
  - Increase network size by n/p where n is number of hidden units in original, p is probability of dropout
- Dropout slows down error convergence
  - Increase learning rate by 10 to 100
  - Or increase momentum (e.g. from 0.9 to 0.99)
  - These can cause large weight growths, use weight regularization
  - May require more iterations to converge

Preventing Overfitting

- Approach 1 Get more data
  - Always best if possible!
  - If no natural ones, use data augmentation
- Approach 2 Regularization
  - Add regularizer term to the objective function
  - Need to incorporate this in back-propagation
- Approach 3 Choose network structure with the right capacity:
  - Enough to fit the true data variations, but not to fit noise
  - Requires parameter tuning, hard to guess the right size.
- Approach 4 Average many different models
  - Models with different forms to encourage diversity (Dropout)
  - Train on different subsets of data
- Approach 5 Batch Normalization
  - Scaling and shifting of features to reduce overfitting

Batch Normalization

- When we pass a batch of samples through a neural network, there may be limited diversity in the intermediate features produced by the batch, referred to as internal covariate shift
- By normalizing the inputs to each layer, we can improve generalization and reduce overfitting
- The original paper has more than 60000 citations!



Batch Normalization

- For a batch of size m, if the inputs to a layer are $x_1, \ldots, x_m$, then the normalized inputs are:

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}} \tag{3}$$

  where μ and σ are the empirical mean and standard deviation over that batch.

- The output of the batch normalization layer is

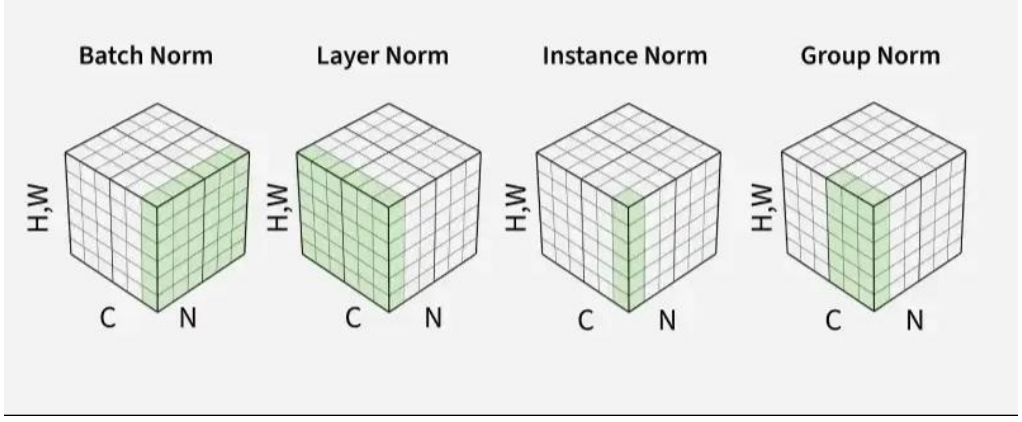
$$y_i = \gamma \hat{x}_i + \beta \tag{4}$$

  where γ and β are parameters of the Batch normalization layer that are trained during backpropagation.

- This transformation is applied to each element of the input to the next layer

Generalizations of Batch Normalization

- Modern neural networks use more general forms of batch normalization
- Instead of just normalizing across the samples of a batch, the normalization can be done across groups of channels in the case of CNNs
- Each channel corresponds to the output of one filter in a CNN.
- In the figure below, N is the batch size, C is the number of channels, and H/W are the height and width of each channel



Preventing Overfitting

- Approach 1 Get more data
  - Always best if possible!
  - If no natural ones, use data augmentation
- Approach 2 Regularization
  - Add regularizer term to the objective function
  - Need to incorporate this in back-propagation
- Approach 3 Choose network structure with the right capacity:
  - Enough to fit the true data variations, but not to fit noise
  - Requires parameter tuning, hard to guess the right size.
- Approach 4 Average many different models
  - Models with different forms to encourage diversity (Dropout)
  - Train on different subsets of data
- Approach 5 Batch Normalization
  - Scaling and shifting of features to reduce overfitting

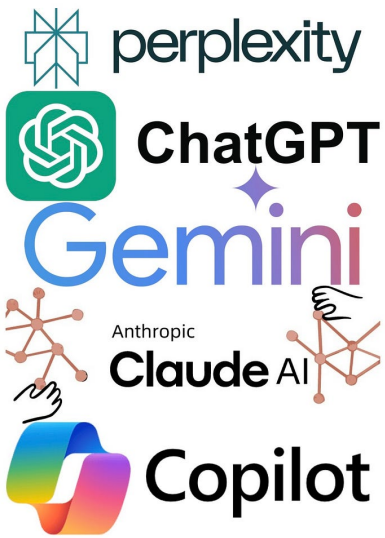
Outline

1. Review: Training a Neural Network: Backpropagatiopn
2. Review: Optimizing SGD Parameters for Faster Convergence
3. Mitigating Overfitting
4. Language Models and RNNs
5. Transformer Language Models

Language Models and RNNs

What is Generative AI?

- So far, we have considered supervised learning tasks where we are given a training dataset of feature-label pairs $(\mathbf{x}_n, y_n)$, for $n = 1, \ldots, N$. Our goal is to learn a function $f(\mathbf{x}_n) \approx y_n$ that maps features to targets/labels
- In generative AI, we do not have explicit labels. Given a sequence of inputs $\mathbf{x}_1, \mathbf{x}_2, \ldots \mathbf{x}_t$ our goal is to predict the next element of the sequence $\mathbf{x}_{t+1}$.



What is Generative AI?

- So far, we have considered supervised learning tasks where we are given a training dataset of feature-label pairs $(\mathbf{x}_n, y_n)$, for $n = 1, \ldots, N$. Our goal is to learn a function $f(\mathbf{x}_n) \approx y_n$ that maps features to targets/labels
- In generative AI, we do not have explicit labels. Given a sequence of inputs $\mathbf{x}_1, \mathbf{x}_2, \ldots \mathbf{x}_t$ our goal is to predict the next element of the sequence $\mathbf{x}_{t+1}$.
- Examples:
  - Next word prediction
  - Text generation given a prompt
  - Machine translation
  - Image/video generation from a description

How does Generative AI work?

- We model the conditional distribution of the next token given the previous tokens:

$$\Pr(\mathbf{x}_{t+1}|\mathbf{x}_t, \ldots \mathbf{x}_1)$$

  using a neural network such as an RNN or transformer

- Then we sample from this probability distribution to generate $\mathbf{x}_{t+1}$

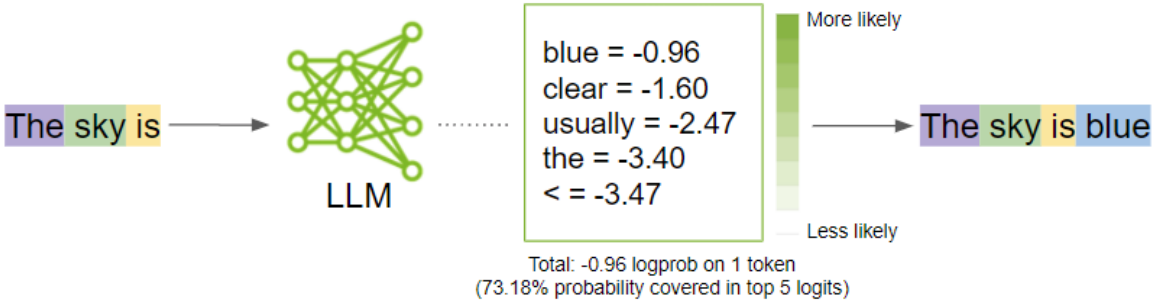
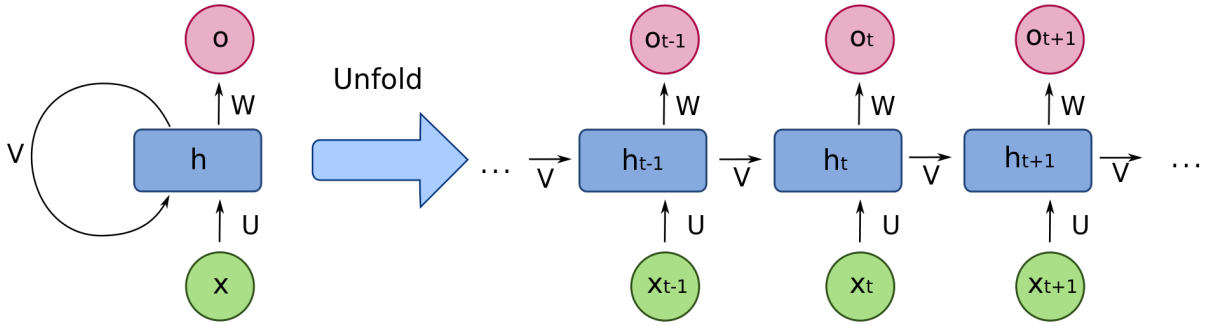


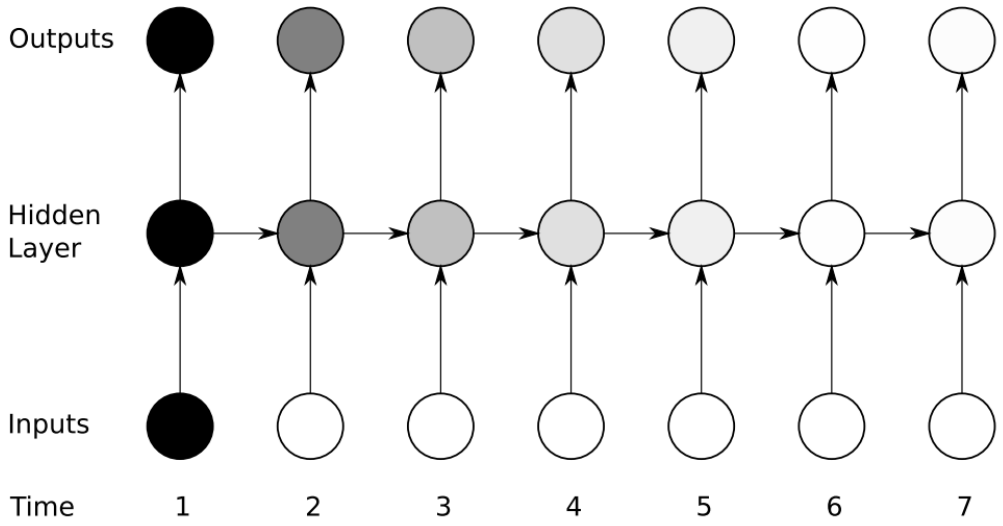
Figure source: NVIDIA technical blog

Recurrent Neural Networks (RNNs)



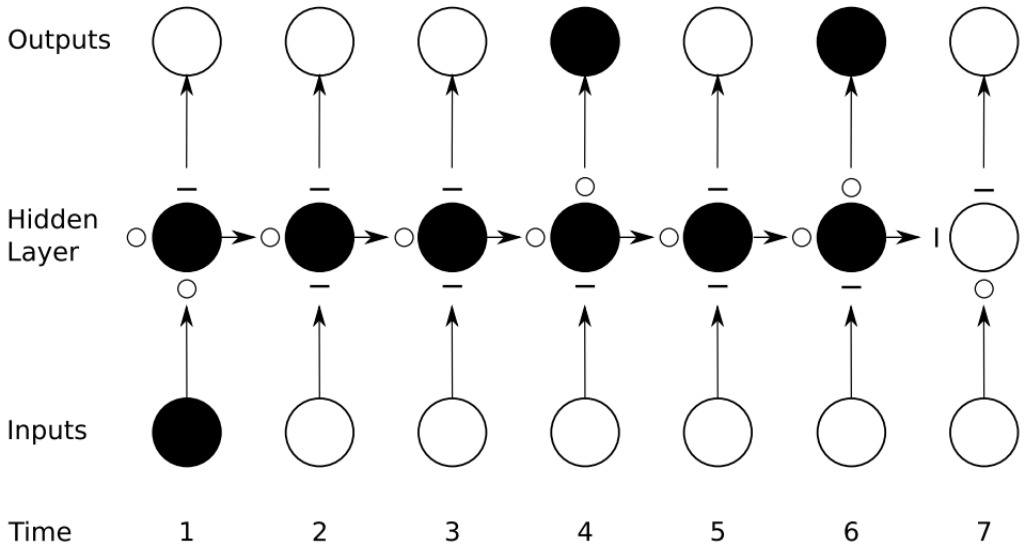
- Precursors to transformers, RNNs were widely used to model temporal or sequential data (e.g., natural language).
- Sequence of hidden states $\mathbf{h}_t$ that depend on the current input $\mathbf{x}_t$ and the previous hidden state $\mathbf{h}_{t-1}$
- Output computation: $\mathbf{o}_t = \psi(\mathbf{W}\mathbf{h}_t + b)$
- Hidden state computation: $\mathbf{h}_t = \phi(\mathbf{V}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t + c)$
- The weight matrices $\mathbf{W}, \mathbf{V}, \mathbf{U}$ and biases b and c are trained using backpropagation on a dataset of sequences of varying lengths
- The predicted output $\mathbf{o}_t$ becomes the next input $\mathbf{x}_{t+1}$

RNNs and Forgetting



- RNNs tend to forget information as they progress forward through the sequence
- This is due to weak or vanishing gradients as we move longer distance through the model
- For a long sentence where the beginning of the sentence has information about the subject, such forgetfulness can be catastrophic

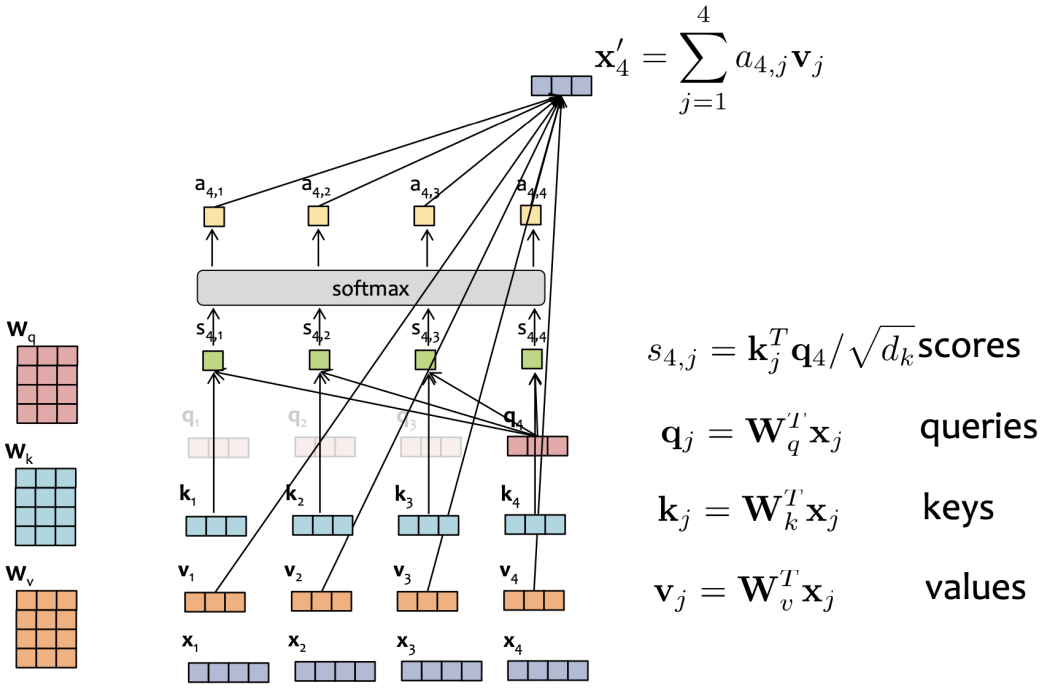
Long and Short-term Memory (LSTM)



- They combat the RNN forgetting issue via gates that decide whether to remember or forget information about the hidden states.
- But they still have drawbacks such as:
  - Difficulty with long-range dependencies (albeit less than RNNs)
  - Even though they solve the vanishing gradient problem, they suffer from exploding gradients
  - Inherently serial computation makes it harder to train

Transformer Language Models

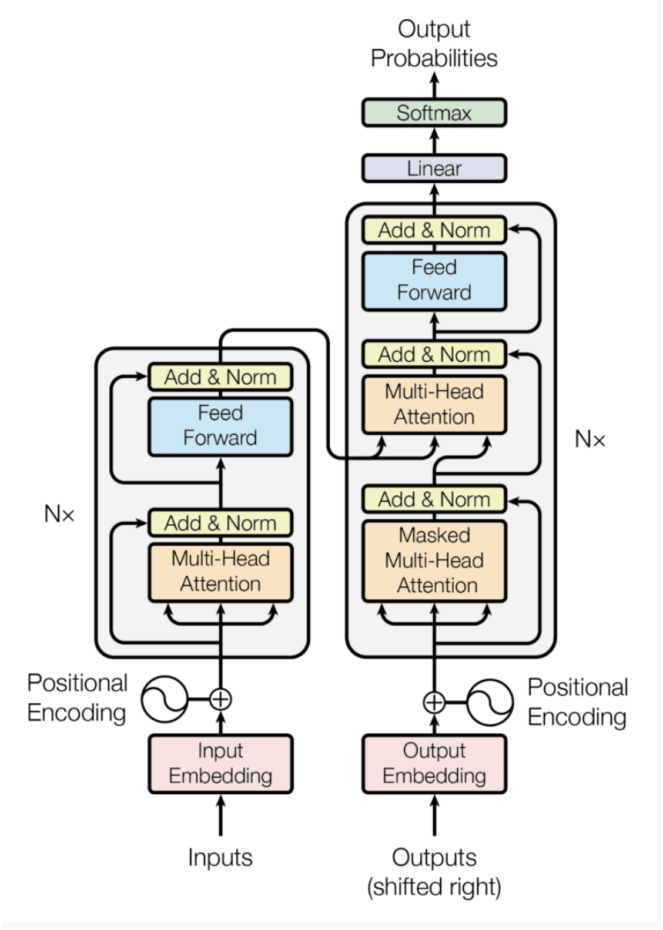
Transformers: Attention Mechanism



- RNNs and LSTM maintain a fixed length hidden state to represent the history of a sequence.
- Instead, the attention mechanism in Transformers looks at all previous token when predicting the next token
- The 'attention' that it pays to each token is computed using the attention mechanism

Transformers

- Encoder-decoder architecture: learn a representation of each input in the sequence, then decode to predict next entry in the output sequence
  - Autoregressive structure: takes previously generated outputs as inputs
  - Attach an attention weight to each entry of each input representation
  - Self-attention between each layer
- Much larger and slower to train, but usually gives good performance



Large Language Models

- "Generative pre-trained transformer" models: *generate* language outputs based on *pre-training* of *transformer*-based architectures on a massive corpus of language data
- Classification task: next-word prediction (run many times)
  - Tokenization: divide text into 'tokens of similar length/information
  - Predict the next token based on the preceding sequence of tokens (typically 1M long)
- Self-/semi-supervised models: generate supervisory signals ("labels") based on output of currently trained model
- Other generative models can generate images, videos, etc. Multi-modal models can, e.g., use a text input to generate an image.

Pre-Training and Finetuning

- Modern deep learning models are too expensive to train from scratch (GPT-4 likely cost millions of dollars to train!) As an example, Llama-7B, 13-B, etc. have billions of parameters.
- Pre-trained foundation models capture essential patterns and can be finetuned to specific datasets
  - Types of language, e.g., coding tools or translation tasks
  - Types of images, e.g., generating images of a certain style
- Foundation models can be trained further on a new dataset
  - Layer freezing or prompt engineering
  - "Warm start" initialization to the usual SGD-based training steps
- Prune, quantize, or compress foundation models to fit them or train them on smaller devices.

Summary

You should know:

- Backpropagation in NNs and optimization algorithms
- How to avoid overfitting in neural networks using Dropout and Batch Normalization
- Language models and RNNs
- Transformer language models