CS 182/282A: Designing, Visualizing and Understanding Deep Neural Networks **Spring 2019**

Section 4: RNNs, Visualizing Networks, Semantic Models for Text,

Notes by: David Chan

4.1 Course Logistics

- You should have heard back about both midterms and project proposals, if not, please talk to one of us ASAP. You should have received an email about checking in with one of us this week to talk about projects, if not, again, please let us know ASAP.
- Midterm re-grade requests are open until the 18th of March.

4.2 Recurrent Neural Networks

The world is full of sequential information. From videos, to language modeling, to time series data, sequences are everywhere. We would like to think about ways to model these sequences using deep neural networks. There are a few major types of tasks that we'd like to solve with sequence models: **One to one** problems take a single input x and produce a single output y. Examples of one to one problems include classification (takes an image as input, and produces a class label as output) and semantic segmentation (image as input, segmentation mask as output). **One to many** models take a single input, and produce a sequence of output. And example of this type of problem is image captioning, which takes a single image as input, and produces a caption (a sequence of words) as an output. **Many to many** tasks are a particularly exciting field - they take sequences of inputs and produce sequences of outputs. One major example of this kind of problem is language translation (sequence of words in one language to sequence of words in another).

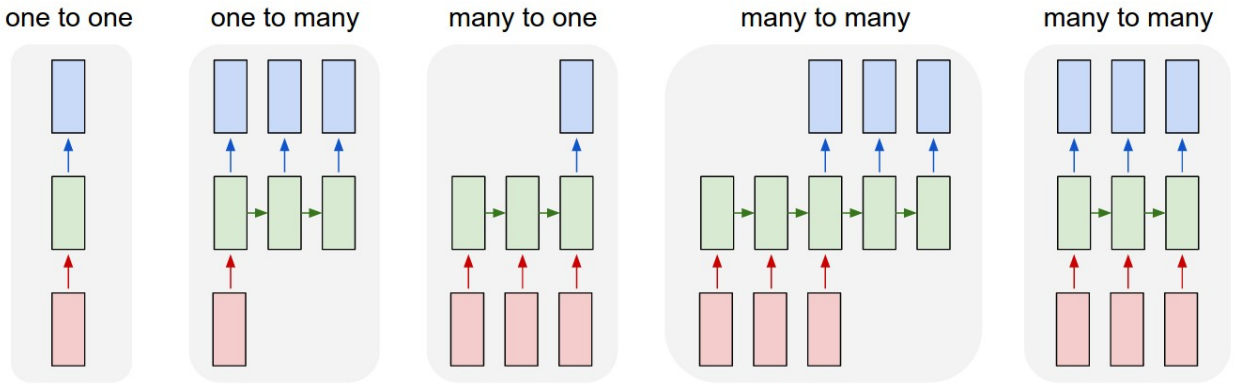
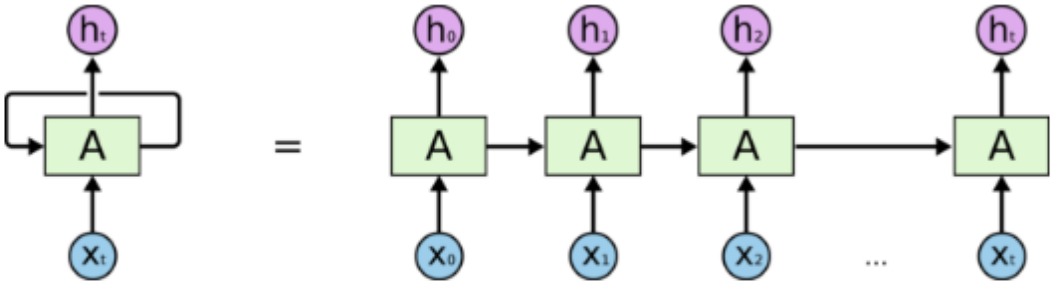


Figure 4.1: Types of problems we would like to solve using sequential models. Figure from `http://karpathy.github.io/2015/05/21/rnn-effectiveness/`

Humans dont start their thinking from scratch every second. As you read this essay, you understand each word based on your understanding of previous words. You dont throw everything away and start thinking from scratch again. Your thoughts have persistence. Traditional neural networks cant do this, and it seems like a major shortcoming. For example, imagine you want to classify what kind of event is happening at every point in a movie. Its unclear how a traditional neural network could use its reasoning about previous events in the film to inform later ones. Recurrent neural networks address this issue. They are networks with loops in them, allowing information to persist. They do this by adding "recurrent connections" to the network, which allow us to allow information to propagate from "the past" (things earlier in the sequence) to the future (or things later in the sequence).



An unrolled recurrent neural network.

Figure 4.2: An example of a generic recurrent neural network. This shows how to "unroll" a network through time - instead of thinking about sequence modeling as a single network with shared weights Figure from `https://medium.com/explore-artificial-intelligence/an-introduction-to-recurrent-neural-networks-72c97bf0912`

4.2.1 Vanilla RNN

For the following section, we'll use the notation that the input is a sequence $x_t, t \in 1...T \in \mathbb{R}^k$, and $y_t, t \in 1...T \in \mathbb{R}^m$ is the sequential output of the network. The classic "vanilla" RNN is made up a node that computes the following:

$$h_t = tanh(W_{hh}h_{t-1} + W_{xh}x_t) \tag{4.1}$$

$$y_t = W_{hy}h_t \tag{4.2}$$

In the vanilla RNN, we can see that at each timestep we compute the update to a hidden state "h" based on the input at the current time, and produce an output which is a projection of the hidden state. To compute the forward (and backwards) passes of the network, we have to "unroll" the network, as shown in Figure 4.2. We can see that this "unrolling" process creates a very deep feed forward network (a network which is similarly deep to the length of the input sequence). Our gradient is computed by taking the loss at each time-step of the output.

The major issue with the vanilla RNN is that it suffers from vanishing/exploding gradients. Because at each timestep the h_t value is multiplied by W, at the last timestep, the value of h_t is effectively multiplied by W^T. This means that depending on the spectral radius of the matrix W, the gradients of the loss with respect to W may become very large or very small as they pass back down the unrolled network.

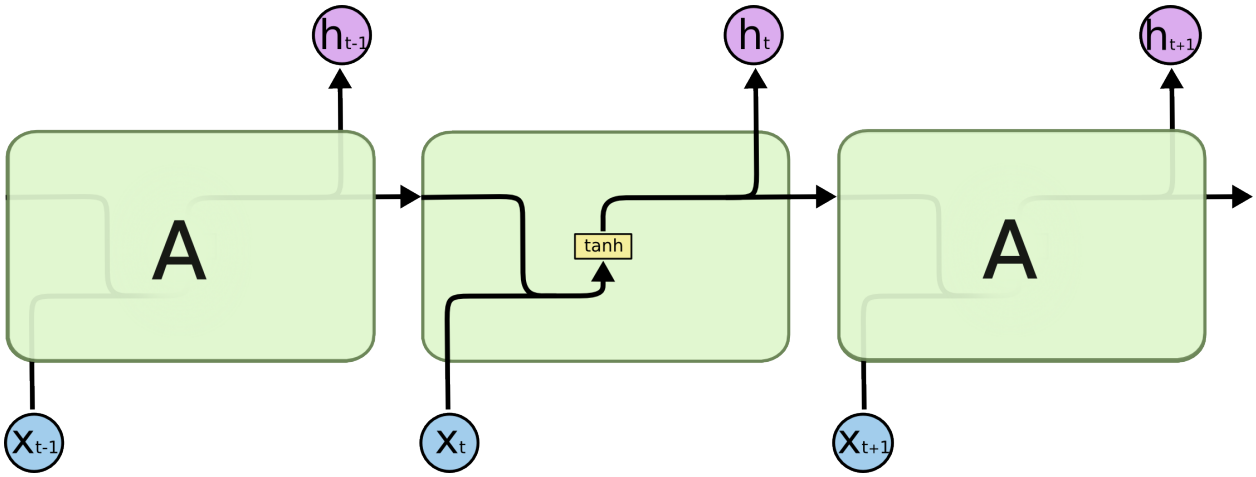


Figure 4.3: A simple RNN cell. As we can see by the arrows, we only pass a single hidden state from time $t-1$ to time t. Figure from `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`

4.2.2 LSTM

To address the probelm of vanishing and exploding gradients, we designed a new kind of recurrent cell - the LSTM cell (standing for "long short term memory"). The layout of the cell is shown as an overview in in Figure 4.4. The LSTM has two states (tensors) which are passed between timesteps, a "Cell memory" C in addition to the hidden state h. The LSTM update is calculated as follows:

$$f_t = \sigma(W_f \cdot [h_t - 1, x_t] + b_f) \tag{4.3}$$
$$i_t = \sigma(W_i \cdot [h_t - 1, x_t] + b_i) \tag{4.4}$$
$$o_t = \sigma(W_o \cdot [h_t - 1, x_t] + b_o) \tag{4.5}$$
$$\hat{C}_t = tanh(W_C \cdot [h_t - 1, x_t] + b_C) \tag{4.6}$$
$$C_t = f_t \circ C_{t-1} + i_t \circ \hat{C}_t \tag{4.7}$$
$$h_t = o_t \circ tanh(C_t) \tag{4.8}$$
$$y_t = h_t \tag{4.9}$$
$$\tag{4.10}$$

The update function is rather complex, but it makes a lot of sense when looking at it in the context of the cell state C as a "memory". First, we compute the value f_t, which we call the "forget" gate. Because of the sigmoid activation function, f_t is between 0 and 1, and the first thing we do is multiply the previous memory times this value. Intuitively, if f_1 is 1, we "remember" the previous state, or if f_t is 0, we forget it. Next, we compute i_t, which we consider as the "input/update" gate. The update gate gets added to the memory cell, so it takes information from the current input x_t and adds it to the memory. Finally, the output gate o_t controls the output of the network, the value that gets passed on to the next cell.

Because of the "forget" gate, the gradients are controlled:

$$\frac{\partial}{\partial c_{t-1}} = \frac{\partial}{\partial c_{t-1}} + f_t \circ \frac{\partial L}{\partial c_t} \tag{4.11}$$

If f_t is the rate at which you want the neural network to "forget" it's past memory, then the error signal is propagated correctly to the next time step. Many works refer to this process as the *linear carousel* which prevents the vanishing of gradients through many time steps.

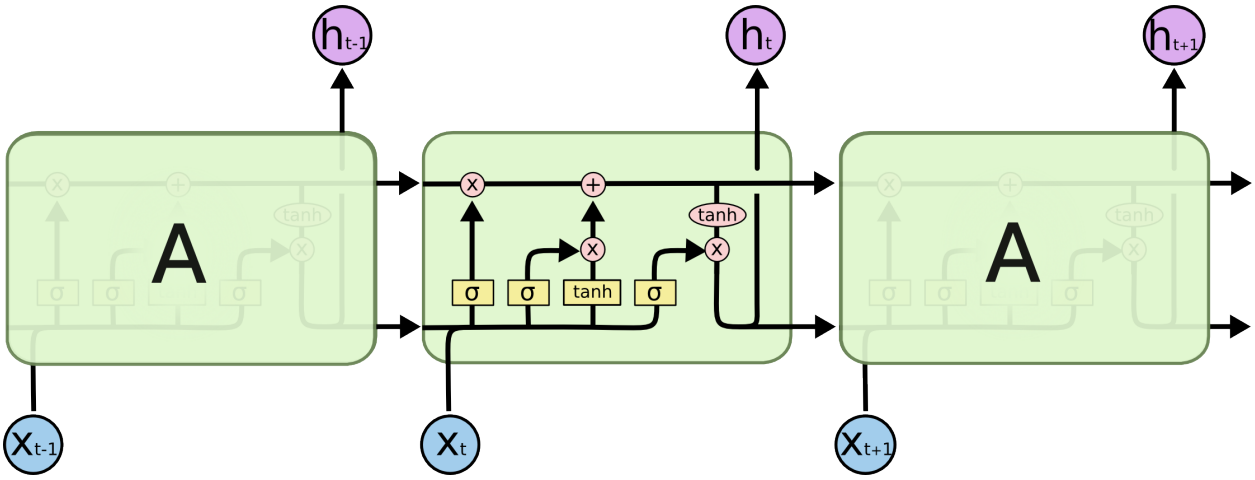


Figure 4.4: An overview of the LSTM cell. Figure from `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`

4.3 Visualizing Deep Neural Networks

Neural networks are increasingly seen as "mysterious" due to the fact that they learn complex high dimensional functions of the input. In many cases, we would like to peer within the black box of the neural network, and attempt to decipher the function that we have learned. There are several ways of doing this, but one of the most common is "Activation Maximization" which we will talk about in more detail here.

4.3.1 Activation Maximization

Consider an image classification task. For an input X, we predict an output vector $y \in \mathbb{R}^C$ where C is the number of classes. These are the "logits" of the network, and to classify, we take the softmax, and pick the highest value. Thus, our prediction is:

$$pred = \text{argmax} \quad softmax(\hat{f}(X, \theta)) \tag{4.12}$$

where $\hat{f}$ is our classifier, θ are our weights, and X is the input. Traditionally, to train the network, we compute the gradients of the loss with respect to the parameters: $\partial L / \partial \theta$. **But what if we want to find the most probable image?** To do this, we can take the derivative of the logits with respect to X, the input. If we then attempt to maximize a logit by making gradient steps on the input image, we will end up with an image which maximizes the probability corresponding to that logit. Because there may be many local minima in this space, we do this many times, with many random initializations of the image (many random starting points) to increase the diversity of the possible samples.

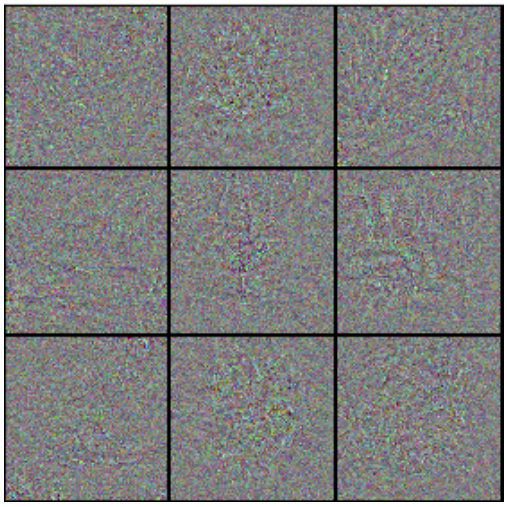


Figure 4.5: ImageNet activation maximization sample from ResNet50 with no regularization or initialization tricks. Notice that we get a random, and rather uninterpretable, value.

The reason that this activation maximization can sometimes not lead to the right value, is a bit subtle: *the space which we are optimizing is not the space of interpretable images*. That is, we're not actually finding maximally activating images *that could come from the real world*, but maximally activating images that come from *pixel space*. To solve this problem we can design a regularization function which acts as an auxiliary loss to our optimization which penalizes unnatural images from occurring.

The right choice of $R(x)$ means everything to activation maximization. For example, total-variation regularization (which minimizes a variation of local pixels) gives us much more interpretable activation maps (as shown in the figure below).

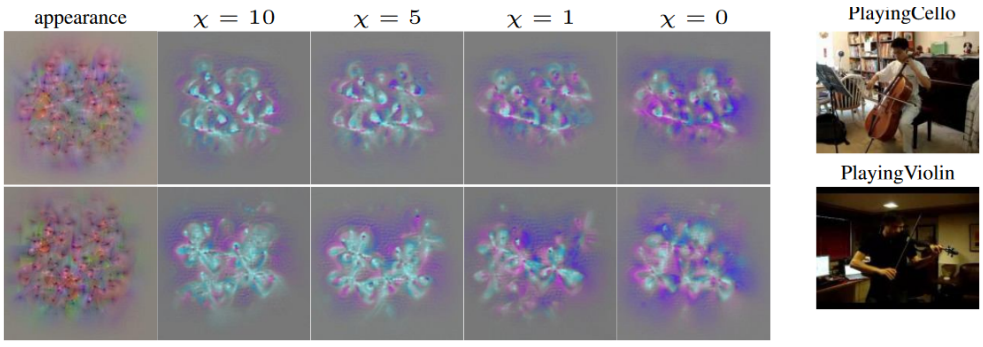


Figure 4.6: Activations on an activity recognition dataset for a network trained to distinguish between playing the cello and violin. Notice how by altering the regularization technique, you can get more and less interpretable images. For more on the total variation regularization, see the paper: `https://arxiv.org/pdf/1801.01415.pdf`

4.4 Semantic Text Models

4.4.1 Vector Embedding of Words

For neural models to act on natural language, we need a way of encoding the language into vectors which the network can process. This section explains a few of those methods.

4.4.1.1 Bag Of Words

The most common solution was to pick a vocab, and encode a word as a one hot vector in that vocab. For example, if we had a small vocab of ”Hello”,”World” and ”Foo”, we might encode the word ”Hello” as the vector $(1,0,0)$ and the word ”World” as the vector $(0,1,0)$ etc. To encode a full sentence, we would just sum these vectors for the whole sentence. This gives us ”token counts” as the input vector - the sentence ”Hello world” becomes $(1,1,0)$ in our example vocabulary.

There are two major issues that we are going to talk about when it comes to bag of words encoding/decoding. One issue with the back of words embedding is that it treats words as equally similar or dissimilar (that is, the L2 distance between two embedded words is the same, no matter what those words are). The other major issue is that it completely ignores word ordering and semantics - that is the sentence ”Dog bites man” has the same encoding as the sentence ”man bites dog” - which is not exactly encouraging.

4.4.2 Word Similarity: Context Embedding

To address the issue of words being embedded at equal distances, we use the idea that *similar words are used in similar contexts*. By counting the words that occur nearby the word, and using those counts instead of the one-hot vector, we get a better ”contextual” representation of our words. Unfortunately, the dimension of these vectors is rather high, so we use PCA or other dimensionality reduction techniques to reduce the dimension of the vectors.

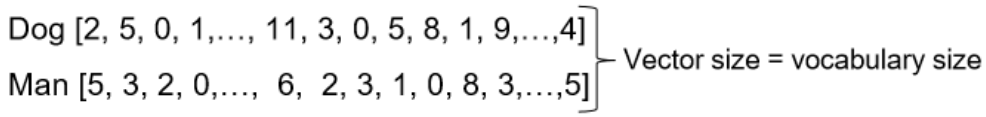


Figure 4.7: Example contextural embeddings of ”Dog” and ”Man”. The counts of words around the word ”Dog” stand in as a meaning for the word ”Dog” and the same for ”Man”. This is much more effective than one-hot encoding, as two vectors for words in similar contexts will have very high inner products (very small cosine distance), and thus addresses the issue of equal distance between each embedding word that is present in the one-hot metric.

4.4.3 Other embedding techniques

LSA: Latent Semantic Analysis. First computes the bag of words matrix for the corpus, and then factorizes it using SVD. These factors encode ”document contexts”, so by using the matrix V of factors from the SVD, we can encode documents x using Vx. SVD is an auto-encoding method, which can be represented

using a linear auto-encoder.

Word2Vec: Word2vec is a two-layer neural net that processes text. Its input is a text corpus and its output is a set of vectors: feature vectors for words in that corpus. There are two ways of building a network show in figure 4.8. The continuous bag of words (CBOW) variant takes the context, and attempts to produce an encoding of the word based on its context. it does this by producing encoding the context words into an N dimensional hidden layer, which is then used to predict the output word. The skip-gram method takes the target word, and tries to reconstruct the context of the word. These models can both be trained using a classification (cross-entropy) loss, and are data-driven ways of learning text embeddings.

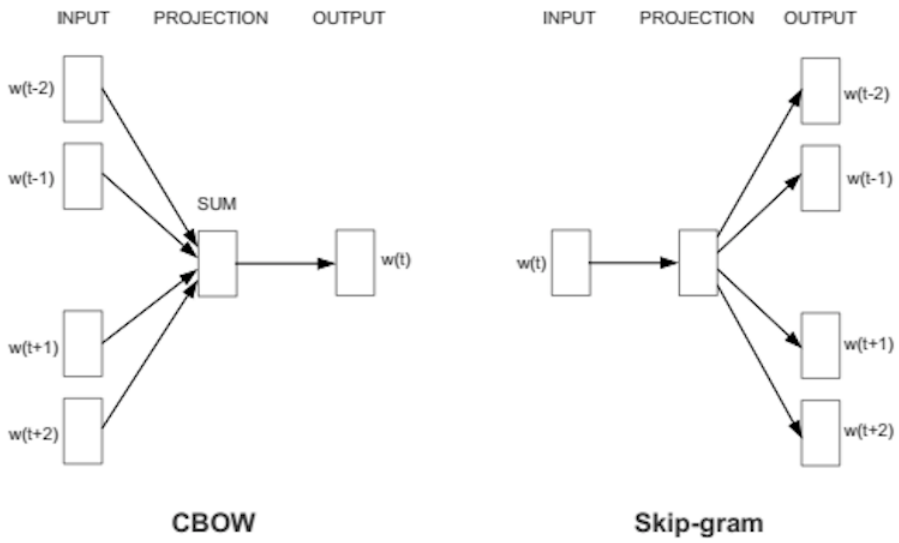


Figure 4.8: Diagram of the word2vec model. By training this like an auto-encoder, we can use the latent dimension features as an encoding of the target word.

Co-occurance Models: A Words co-occurrence matrix describes how words occur together, and in turn captures the relationships between words. The Words co-occurrence matrix is computed by counting how many times two or more words occur together in a given corpus, and putting this into a square matrix. We can decompose this model using SVD or PCA to get factors/principal components which can be used to reduce the dimensionality of the vocab, and embed words similar to the way that it is done in LSA.

GloVe: GloVe is an unsupervised learning algorithm for obtaining vector representations for words. Training is performed on aggregated global word-word co-occurrence statistics from a corpus, and the resulting representations showcase interesting linear substructures of the word vector space. Glove is a learned variant of the co-occurance models, where a neural network is used for dimensionality reduction instead of a fixed SVM/PCA method.

ELMo: Similar to the Word2Vec model, except use the hidden state of an LSTM trained on the language modeling task.

4.5 Hard Midterm Questions

4.5.1 1 (t)

When using activation maximization to visualize a neuron, why is the initialization of the image important?

As we already discussed in this section, activation maximization is **a gradient method** designed to find the image with the highest activation for a particular neuron. Because the neural network is a non-linear function approximator, there are a number of possible local minima, and depending on our initialization (or our starting point), we will get a different local minima (or highly activating image). This contributes to a diversity of visualizations. This is useful, because many different data distributions are multi-modal (there are many types of cat), so we want a representative sample of the diversity of the means in the data distribution.

4.5.2 2 (a)

A soft-margin SVM minimizes the following loss function: $L = \sum_{i=1}^{n} \max(0, 1 - y_i f(x_i)) + \lambda ||w||^2$**. How does the loss vary as a function of** $||w||$ **ignoring the regularization term**

If we assume that the weight is scaled by a scalar $w = \alpha v$, we can substitute it into the loss formula:

$$L = \sum_{i=1}^{n} \max(0, 1 - y_i(b + \alpha v^T x_i) \tag{4.13}$$

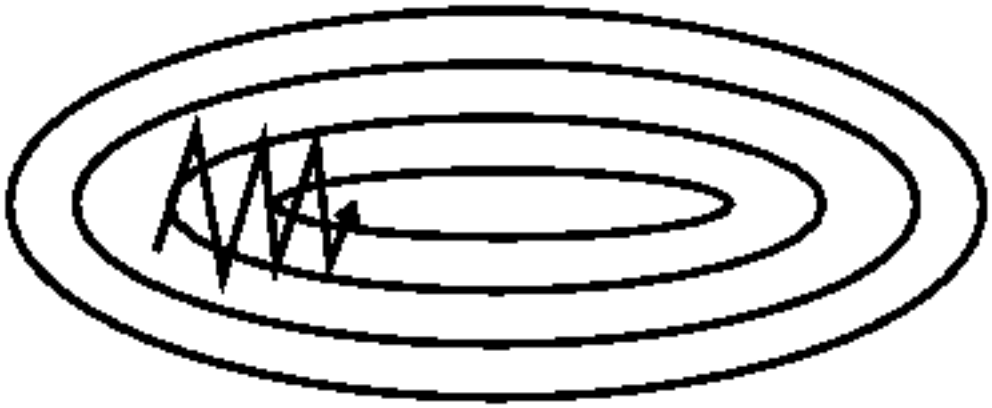
For $\alpha = 0$, the loss is:

$$L = \sum_{i=1}^{n} \max(0, 1 - y_i(b) \tag{4.14}$$

Now let's consider what happens as α increases. As α increases away from 0, we can see that the loss will increase or decrease linearly, depending on if the model is classifying things correctly, or incorrectly. Notice that if the model is classifying correctly - then increasing the magnitude of the weight matrix will decrease the loss. If the model is classifying incorrectly on the other hand, increasing the magnitude of the weight matrix will increase the loss.

4.5.3 3

This question is a bit tricky to parse, but once you parse it, you can see what the question is asking, and what happens for each of the example gradient descent methods. In this question F has a spherical loss, however the function G scales the loss so that it is ellipsoidal, shown in the figure below:



In this figure, we can see that vanilla SGD will perform poorly, because of the high oscillation in the gradients which is caused by the ellipsoidal shape. Thus, the loss will be higher, and it will take longer converge, as the gradients to not point directly at the optimum, but instead can be almost orthogonal (depending on the dimensions). At high learning rates, vanilla SGD will oscillate.

SGD+Momentum should do a slightly better job optimizing G than the vanilla version, as at high enough learning rates the gradients will cancel each other out as they oscillate, learning to a net gradient which is close to optimal. The time to reach the final loss might be similar to F, and the loss should be roughly the same.

For RMSProp the gradients are individually scaled, so the losses have a unit-ball shape. This means that the transformation that we made can be ”undone” by RMSProp, and RMSProp on G will behave similarly to standard SGD on F.

ADAM is similar, and scales the gradients by the inverse gradient magnitude. Because G is a diagonally scaled version of F, ADAM on G will have similar gradients to SGD on F, and have similar convergence properties.

4.5.4 5 (c)

How does a bottleneck layer impact the overall network accuracy?

Accuracy may increase or decrease. Non-bottleneck layers have more parameters and therefore potentially lower bias if there is enough training data. On the other hand, the lower parameter count for bottleneck reduces the risk of over-fitting on modest-sized training datasets.